



---

# Понимание языка ассемблера

(Reverse Engineering для начинающих)

Почему два названия? Читайте здесь: ?? (стр. ??).

Денис Юричев

[мои адреса](#)



©2013-2022, Денис Юричев.

Это произведение доступно по лицензии Creative Commons «Attribution-ShareAlike 4.0 International» (CC BY-SA 4.0). Чтобы увидеть копию этой лицензии, посетите

<https://creativecommons.org/licenses/by-sa/4.0/>.

Версия этого текста (26 марта 2023 г.).

Самая новая версия текста (а также англоязычная версия) доступна на сайте <https://beginners.re/>.

## Нужны переводчики!

Возможно, вы захотите мне помочь с переводом этой работы на другие языки, кроме английского и русского. Просто пришлите мне любой фрагмент переведенного текста (не важно, насколько короткий), и я добавлю его в исходный код на LaTeX.

Не спрашивайте, нужно ли переводить. Просто делайте хоть что-нибудь. Я уже перестал отвечать на емейлы вроде “что нужно сделать?”

[Также, прочитайте это.](#)

Посмотреть статистику языков можно прямо здесь: <https://beginners.re/>.

Скорость не важна, потому что это опен-сорсный проект все-таки. Ваше имя будет указано в числе участников проекта. Корейский, китайский и персидский языки зарезервированы издателями. Английскую и русскую версии я делаю сам, но английский у меня все еще ужасный, так что я буду очень признателен за коррективы, и т. д. Даже мой русский несовершенный, так что я благодарен за коррективы и русского текста!

Не стесняйтесь писать мне: [мои адреса](#).

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

# **Краткое оглавление**

## **Оглавление**

---

## Предисловие

### Почему два названия?

В 2014-2018 книга называлась “Reverse Engineering для начинающих”, но я всегда подозревал что это слишком сужает аудиторию.

Люди от инфобезопасности знают о “reverse engineering”, но я от них редко слышу слово “ассемблер”.

Точно также, термин “reverse engineering” слишком незнакомый для общей аудитории программистов, но они знают про “ассемблер”.

В июле 2018, для эксперимента, я заменил название на “Assembly Language for Beginners” и запостил ссылку на сайт Hacker News<sup>1</sup>, и книгу приняли, в общем, хорошо.

Так что, пусть так и будет, у книги будет два названия.

Хотя, я поменял второе название на “Understanding Assembly Language” (“Понимание языка ассемблера”), потому что кто-то уже написал книгу “Assembly Language for Beginners”. Также, люди говорят что “для начинающих” уже звучит немного саркастично для книги объемом в ~1000 страниц.

Книги отличаются только названием, именем файла (UAL-XX.pdf и RE4B-XX.pdf), URL-ом и парой первых страниц.

### О reverse engineering

У термина «[reverse engineering](#)» несколько популярных значений: 1) исследование скомпилированных программ; 2) сканирование трехмерной модели для последующего копирования; 3) восстановление структуры СУБД.

Настоящая книга связана с первым значением.

### Желательные знания перед началом чтения

Очень желательно базовое знание **PLI**<sup>2</sup> Си. Рекомендуемые материалы: ?? (стр. ??).

### Упражнения и задачи

...все перемещены на отдельный сайт: <http://challenges.re>.

### Отзывы об этой книге

<https://beginners.re/#praise>.

<sup>1</sup><https://news.ycombinator.com/item?id=17549050>

<sup>2</sup>PLI



---

## Университеты

Эта книга рекомендуется по крайней мере в этих университетах: <https://beginners.re/#uni>.

## Благодарности

Тем, кто много помогал мне отвечая на массу вопросов: SkullC0DEr.

Тем, кто присылал замечания об ошибках и неточностях: Александр Лысенко, Федерико Рамондино, Марк Уилсон, Разихова Мейрамгуль Кайратовна, Анатолий Прокофьев, Костя Бегунец, Валентин “netch” Нечаев, Александр Плахов, Артем Метла, Александр Ястребов, Влад Головкин<sup>3</sup>, Евгений Прошин, Александр Мясников, Алексей Третьяков, Олег Песков, Павел Шахов, Zhu Ruijin, Changmin Heo, Vitor Vidal, Stijn Crevits, Jean-Gregoire Foulon<sup>4</sup>, Ben L., Etienne Khan, Norbert Szetei<sup>5</sup>, Marc Remy, Michael Hansen, Derk Barten, The Renaissance<sup>6</sup>, Hugo Chan, Emil Mursalimov, Tanner Hoke, Tan90909090@GitHub, Ole Petter Orhagen, Sourav Punoriyar, Vitor Oliveira, Alexis Ehret, Maxim Shlochiski, Greg Paton, Pierrick Lebourgeois, Abdullah Alomair, Bobby Battista..

Просто помогли разными способами: Андрей Зубинский, Arnaud Patard (rtp на #debian-arm IRC), noshadow на #gcc IRC, Александр Автаев, Mohsen Mostafa Jokar, Пётр Советов, Миша “tiphareth” Вербицкий.

Переводчикам на китайский язык: Antiy Labs ([antiy.cn](http://antiy.cn)), Archer.

Переводчику на корейский язык: Byungho Min.

Переводчику на голландский язык: Cedric Sambre (AKA Midas).

Переводчикам на испанский язык: Diego Boy, Luis Alberto Espinosa Calvo, Fernando Guida, Diogo Mussi, Patricio Galdames, Emiliano Estevarena.

Переводчикам на португальский язык: Thales Stevan de A. Gois, Diogo Mussi, Luiz Filipe, Primo David Santini.

Переводчикам на итальянский язык: Federico Ramondino<sup>7</sup>, Paolo Stivanin<sup>8</sup>, twyK, Fabrizio Bertone, Matteo Sticco, Marco Negro<sup>9</sup>, bluepulsar.

Переводчикам на французский язык: Florent Besnard<sup>10</sup>, Marc Remy<sup>11</sup>, Baudouin Landais, Téo Dacquet<sup>12</sup>, BlueSkeye@GitHub<sup>13</sup>.

---

<sup>3</sup>[goto-vlad@github](mailto:goto-vlad@github)

<sup>4</sup><https://github.com/pixjuan>

<sup>5</sup><https://github.com/73696e65>

<sup>6</sup><https://github.com/TheRenaissance>

<sup>7</sup><https://github.com/pinkrab>

<sup>8</sup><https://github.com/paolostivanin>

<sup>9</sup><https://github.com/Internaut401>

<sup>10</sup><https://github.com/besnardf>

<sup>11</sup><https://github.com/mremy>

<sup>12</sup><https://github.com/T30rix>

<sup>13</sup><https://github.com/BlueSkeye>

Переводчикам на немецкий язык: Dennis Siekmeier<sup>14</sup>, Julius Angres<sup>15</sup>, Dirk Loser<sup>16</sup>, Clemens Tamme, Philipp Schweinzer.

Переводчикам на польский язык: Kateryna Rozanova, Aleksander Mistewicz, Wiktor Lewicka, Marcin Sokołowski.

Переводчикам на японский язык: shmz@github<sup>17</sup>, 4ryuJP@github<sup>18</sup>.

Корректорам: Владимир Ботов, Андрей Бражук, Марк “Logxen” Купер, Yuan Jochen Kang, Mal Malakov, Lewis Porter, Jarle Thorsen, Hong Xie.

Васил Колев<sup>19</sup> сделал очень много исправлений и указал на многие ошибки.

И ещё всем тем на github.com кто присылал замечания и исправления.

Было использовано множество пакетов  $\LaTeX$ . Их авторов я также хотел бы поблагодарить.

## Жертвователи

Тем, кто поддерживал меня во время написания этой книги:

2 \* Oleg Vygovsky (50+100 UAH), Daniel Bilar (\$50), James Truscott (\$4.5), Luis Rocha (\$63), Joris van de Vis (\$127), Richard S Shultz (\$20), Jang Minchang (\$20), Shade Atlas (5 AUD), Yao Xiao (\$10), Pawel Szczur (40 CHF), Justin Simms (\$20), Shawn the R0ck (\$27), Ki Chan Ahn (\$50), Triop AB (100 SEK), Ange Albertini (€10+50), Sergey Lukianov (300 RUR), Ludvig Gislason (200 SEK), Gérard Labadie (€40), Sergey Volchkov (10 AUD), Vankayala Vigneswararao (\$50), Philippe Teuwen (\$4), Martin Haerberli (\$10), Victor Cazacov (€5), Tobias Sturzenegger (10 CHF), Sonny Thai (\$15), Bayna AlZaabi (\$75), Redfive B.V. (€25), Joona Oskari Heikkilä (€5), Marshall Bishop (\$50), Nicolas Werner (€12), Jeremy Brown (\$100), Alexandre Borges (\$25), Vladimir Dikovski (€50), Jiarui Hong (100.00 SEK), Jim Di (500 RUR), Tan Vincent (\$30), Sri Harsha Kandrakota (10 AUD), Pillay Harish (10 SGD), Timur Valiev (230 RUR), Carlos Garcia Prado (€10), Salikov Alexander (500 RUR), Oliver Whitehouse (30 GBP), Katy Moe (\$14), Maxim Dyakonov (\$3), Sebastian Aguilera (€20), Hans-Martin Münch (€15), Jarle Thorsen (100 NOK), Vitaly Osipov (\$100), Yuri Romanov (1000 RUR), Aliaksandr Autayeu (€10), Tudor Azoitei (\$40), Z0vsky (€10), Yu Dai (\$10), Anonymous (\$15), Vladislav Chelnokov (\$25), Nenad Noveljic (\$50), Ryan Smith (\$25), Andreas Schommer (€5), Nikolay Gavrilov (\$300), Ernesto Bonev Reynoso (\$30).

Огромное спасибо каждому!

## mini-ЧаВО

Q: Эта книга проще/легче других?

<sup>14</sup><https://github.com/DSiekmeier>

<sup>15</sup><https://github.com/JAngres>

<sup>16</sup><https://github.com/PolymathMonkey>

<sup>17</sup><https://github.com/shmz>

<sup>18</sup><https://github.com/4ryuJP>

<sup>19</sup><https://vasil.ludost.net/>

A: Нет, примерно на таком же уровне, как и остальные книги посвященные этой теме.

Q: Мне страшно начинать читать эту книгу, здесь более 1000 страниц. "... для начинающих" в названии звучит слегка саркастично.

A: Основная часть книги это масса разных листингов. И эта книга действительно для начинающих, тут многого (пока) не хватает.

Q: Что необходимо знать перед чтением книги?

A: Желательно иметь базовое понимание Си/Си++.

Q: Должен ли я изучать сразу x86/x64/ARM и MIPS? Это не многовато?

A: Для начала, вы можете читать только о x86/x64, пропуская/пролистывая части о ARM/MIPS.

Q: Возможно ли купить русскую/английскую бумажную книгу?

A: К сожалению нет, пока ни один издатель не заинтересовался в издании русской или английской версии. А пока вы можете распечатать/переплести её в вашем любимом копи-шопе/копи-центре. [https://yurichev.com/news/20200222\\_printed\\_RE4B/](https://yurichev.com/news/20200222_printed_RE4B/).

Q: Существует ли версия epub/mobi?

A: Книга очень сильно завязана на специфические для TeX/LaTeX хаки, поэтому преобразование в HTML (epub/mobi это набор HTML) легким не будет.

Q: Зачем в наше время нужно изучать язык ассемблера?

A: Если вы не разработчик **OS!**<sup>20</sup>, вам наверное не нужно писать на ассемблере: современные компиляторы (2010-ые) оптимизируют код намного лучше человека<sup>21</sup>.

К тому же, современные **CPU!**<sup>22</sup> это крайне сложные устройства и знание ассемблера вряд ли поможет узнать их внутренности.

Но все-таки остается по крайней мере две области, где знание ассемблера может хорошо помочь: 1) исследование malware (зловредов) с целью анализа; 2) лучшее понимание вашего скомпилированного кода в процессе отладки. Таким образом, эта книга предназначена для тех, кто хочет скорее понимать ассемблер, нежели писать на нем, и вот почему здесь масса примеров, связанных с результатами работы компиляторов.

Q: Я кликнул на ссылку внутри PDF-документа, как теперь вернуться назад?

A: В Adobe Acrobat Reader нажмите сочетание Alt+LeftArrow. В Evince кликните на "<".

Q: Могу ли я распечатать эту книгу? Использовать её для обучения?

---

<sup>20</sup>**OS!**

<sup>21</sup>Очень хороший текст на эту тему: [Agnér Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)]

<sup>22</sup>**CPU!**

A: Конечно, поэтому книга и лицензирована под лицензией Creative Commons (CC BY-SA 4.0).

Q: Почему эта книга бесплатная? Вы проделали большую работу. Это подозрительно, как и многие другие бесплатные вещи.

A: По моему опыту, авторы технической литературы делают это, в основном ради саморекламы. Такой работой заработать приличные деньги невозможно.

Q: Как можно найти работу reverse engineer-а?

A: На reddit, посвященному RE<sup>23</sup>, время от времени бывают hiring thread. Посмотрите там.

В смежном субреддите «netsec» имеется похожий тред.

Q: У меня есть вопрос...

A: Напишите мне его емейлом ([мои адреса](#)).

## О переводе на корейский язык

В январе 2015, издательство Asorn в Южной Корее сделало много работы в переводе и издании моей книги (по состоянию на август 2014) на корейский язык. Она теперь доступна на [их сайте](#).

Переводил ByungHo Min ([twitter/tais9](#)). Обложку нарисовал мой хороший знакомый художник Андрей Нечаевский [facebook/andydinka](#). Они также имеют права на издание книги на корейском языке. Так что если вы хотите иметь *настоящую* книгу на полке на корейском языке и хотите поддержать мою работу, вы можете купить её.

## О переводе на персидский язык (фарси)

В 2016 году книга была переведена Mohsen Mostafa Jokar (который также известен иранскому сообществу по переводу руководства Radare<sup>24</sup>). Книга доступна на сайте издательства<sup>25</sup> (Pendare Pars).

Первые 40 страниц: <https://beginners.re/farsi.pdf>.

Регистрация книги в Национальной Библиотеке Ирана: <http://opac.nlai.ir/opac-prod/bibliographic/4473995>.

## О переводе на китайский язык

В апреле 2017, перевод на китайский был закончен китайским издательством RTPress. Они также имеют права на издание книги на китайском языке.

Она доступна для заказа здесь: <http://www.epubit.com.cn/book/details/4174>. Что-то вроде рецензии и история о переводе: <http://www.cptoday.cn/news/detail/3155>.

---

<sup>23</sup>[reddit.com/r/ReverseEngineering/](http://reddit.com/r/ReverseEngineering/)

<sup>24</sup><http://rada.re/get/radare2book-persian.pdf>

<sup>25</sup><http://goo.gl/2Tzx0H>

Основным переводчиком был Archer, перед которым я теперь в долгу. Он был крайне дотошным (в хорошем смысле) и сообщил о большинстве известных ошибок и баг, что крайне важно для литературы вроде этой книги. Я буду рекомендовать его услуги всем остальным авторам!

Ребята из [Antiy Labs](#) также помогли с переводом. [Здесь предисловие](#) написанное ими.

## Не устарела ли эта книга?

Меня часто спрашивают: Почему в книге RE4B слишком старые версии компиляторов? Не устарела ли сама книга?

Потому что работа над ней [была начата еще в начале 2010](#). (Поэтому местами даже используется MSVC 2008.)

Нет, не устарела, потому что:

- 1) Компиляторы меняются не так уж и быстро, так что код создаваемый MSVC 2010 и самым последним может не так уж и сильно отличаться.
- 2) То, что вы будете реверсить (malware, и тот софт, в котором будете искать уязвимости), понятно дело, не всегда компилируется свежими компиляторами.

На моей памяти, Oracle RDBMS (когда я еще им занимался) собирался старыми версиями Intel C++. Есть [подозрение](#), что последние версии Windows тоже далеко не всегда собираются последними версиями MSVC.

А следовать всем шагам в книге в точности не обязательно, поэтому не так уж и важно использовать те же компиляторы. Пользуйтесь теми компиляторами, которые уже инсталлированы в вашу OS. А также, всегда есть [Compiler Explorer](#).

# Глава 1

## Образцы кода

### 1.1. Метод

Когда автор этой книги учил Си, а затем Си++, он просто писал небольшие фрагменты кода, компилировал и смотрел, что получилось на ассемблере. Так было намного проще понять<sup>1</sup>. Он делал это такое количество раз, что связь между кодом на Си/Си++ и тем, что генерирует компилятор, вбилась в его подсознание достаточно глубоко. После этого не трудно, глядя на код на ассемблере, сразу в общих чертах понимать, что там было написано на Си. Возможно это поможет кому-то ещё.

Иногда здесь используются достаточно древние компиляторы, чтобы получить самый короткий (или простой) фрагмент кода.

Кстати, есть очень неплохой вебсайт где можно делать всё то же самое, с разными компиляторами, вместо того чтобы устанавливать их у себя. Вы можете использовать и его: <http://godbolt.org/>.

### Упражнения

Когда автор этой книги учил ассемблер, он также часто компилировал короткие функции на Си и затем постепенно переписывал их на ассемблер, с целью получить как можно более короткий код. Наверное, этим не стоит заниматься в наше время на практике (потому что конкурировать с современными компиляторами в плане эффективности очень трудно), но это очень хороший способ разобраться в ассемблере лучше. Так что вы можете взять любой фрагмент кода на ассемблере в этой книге и постараться сделать его короче. Но не забывайте о тестировании своих результатов.

---

<sup>1</sup>Честно говоря, он и до сих пор так делает, когда не понимает, как работает некий код. Недавний пример из 2019-го года: `p += p+(i&1)+2;` из SAT-солвера "SAT0W" написанного Д.Кнутом.

## Уровни оптимизации и отладочная информация

Исходный код можно компилировать различными компиляторами с различными уровнями оптимизации. В типичном компиляторе этих уровней около трёх, где нулевой уровень — отключить оптимизацию. Различают также уровни оптимизации кода по размеру и по скорости. Неоптимизирующий компилятор работает быстрее, генерирует более понятный (хотя и более объемный) код. Оптимизирующий компилятор работает медленнее и старается сгенерировать более быстрый (хотя и не обязательно краткий) код. Наряду с уровнями оптимизации компилятор может включать в конечный файл отладочную информацию, производя таким образом код, который легче отлаживать. Одна очень важная черта отладочного кода в том, что он может содержать связи между каждой строкой в исходном коде и адресом в машинном коде. Оптимизирующие компиляторы обычно генерируют код, где целые строки из исходного кода могут быть оптимизированы и не присутствовать в итоговом машинном коде. Практикующий reverse engineer обычно сталкивается с обеими версиями, потому что некоторые разработчики включают оптимизацию, некоторые другие — нет. Вот почему мы постараемся поработать с примерами для обеих версий.

## 1.2. Некоторые базовые понятия

### 1.2.1. Краткое введение в CPU

**CPU!** это устройство исполняющее все программы.

**Немного терминологии:**

**Инструкция** : примитивная команда **CPU!**. Простейшие примеры: перемещение между регистрами, работа с памятью, примитивные арифметические операции. Как правило, каждый **CPU!** имеет свой набор инструкций (**ISA!**<sup>2</sup>).

**Машинный код** : код понимаемый **CPU!**. Каждая инструкция обычно кодируется несколькими байтами.

**Язык ассемблера** : машинный код плюс некоторые расширения, призванные облегчить труд программиста: макросы, имена, и т. д.

**Регистр CPU** : Каждый **CPU!** имеет некоторый фиксированный набор регистров общего назначения (**GPR!**<sup>3</sup>). ≈ 8 в x86, ≈ 16 в x86-64, ≈ 16 в ARM. Проще всего понимать регистр как временную переменную без типа. Можно представить, что вы пишете на **PL!** высокого уровня и у вас только 8 переменных шириной 32 (или 64) бита. Можно сделать очень много используя только их!

Откуда взялась разница между машинным кодом и **PL!** высокого уровня? Ответ в том, что люди и **CPU!**-ы отличаются друг от друга — человеку проще

---

<sup>2</sup>**ISA!**

<sup>3</sup>**GPR!**

писать на **PL!** высокого уровня вроде Си/Си++, Java, Python, а **СРU!** проще работать с абстракциями куда более низкого уровня. Возможно, можно было бы придумать **СРU!** исполняющий код **PL!** высокого уровня, но он был бы значительно сложнее, чем те, что мы имеем сегодня. И наоборот, человеку очень неудобно писать на ассемблере из-за его низкоуровневости, к тому же, крайне трудно обойтись без мелких ошибок. Программа, переводящая код из **PL!** высокого уровня в ассемблер называется *компилятором*<sup>4</sup>.

### Несколько слов о разнице между **ISA!**

x86 всегда был архитектурой с инструкциями переменной длины, так что когда пришла 64-битная эра, расширения x64 не очень сильно повлияли на **ISA!**. ARM это **RISC!**<sup>5</sup>-процессор разработанный с учетом инструкций одинаковой длины, что было некоторым преимуществом в прошлом. Так что в самом начале все инструкции ARM кодировались 4-мя байтами<sup>6</sup>. Это то, что сейчас называется «режим ARM». Потом они подумали, что это не очень экономично. На самом деле, самые используемые инструкции<sup>7</sup> процессора на практике могут быть закодированы с использованием меньшего количества информации. Так что они добавили другую **ISA!** с названием Thumb, где каждая инструкция кодируется всего лишь 2-мя байтами. Теперь это называется «режим Thumb». Но не все инструкции ARM могут быть закодированы в двух байтах, так что набор инструкций Thumb ограниченный. Код, скомпилированный для режима ARM и Thumb может сосуществовать в одной программе. Затем создатели ARM решили, что Thumb можно расширить: так появился Thumb-2 (в ARMv7). Thumb-2 это всё ещё двухбайтные инструкции, но некоторые новые инструкции имеют длину 4 байта. Распространено заблуждение, что Thumb-2 — это смесь ARM и Thumb. Это не верно. Режим Thumb-2 был дополнен до более полной поддержки возможностей процессора и теперь может легко конкурировать с режимом ARM. Основное количество приложений для iPod/iPhone/iPad скомпилировано для набора инструкций Thumb-2, потому что Xcode делает так по умолчанию. Потом появился 64-битный ARM. Это **ISA!** снова с 4-байтными инструкциями, без дополнительного режима Thumb. Но 64-битные требования повлияли на **ISA!**, так что теперь у нас 3 набора инструкций ARM: режим ARM, режим Thumb (включая Thumb-2) и ARM64. Эти наборы инструкций частично пересекаются, но можно сказать, это скорее разные наборы, нежели вариации одного. Следовательно, в этой книге постараемся добавлять фрагменты кода на всех трех ARM **ISA!**. Существует ещё много **RISC! ISA!** с инструкциями фиксированной 32-битной длины — это как минимум MIPS, PowerPC и Alpha AXP.

<sup>4</sup>В более старой русскоязычной литературе также часто встречается термин «транслятор».

<sup>5</sup>**RISC!**

<sup>6</sup>Кстати, инструкции фиксированного размера удобны тем, что всегда можно легко узнать адрес следующей (или предыдущей) инструкции. Эта особенность будет рассмотрена в секции об операторе switch() (?? (стр. ??)).

<sup>7</sup>А это MOV/PUSH/CALL/Jcc



## 1.2.2. Представление чисел

Nowadays octal numbers seem to be used for exactly one purpose—file permissions on POSIX systems—but hexadecimal numbers are widely used to emphasize the bit pattern of a number over its numeric value.

Alan A. A. Donovan, Brian W. Kernighan —  
The Go Programming Language

Люди привыкли к десятичной системе счисления вероятно потому что почти у каждого есть по 10 пальцев. Тем не менее, число 10 не имеет особого значения в науке и математике. Двоичная система естественна для цифровой электроники: 0 означает отсутствие тока в проводе и 1 — его присутствие. 10 в двоичной системе это 2 в десятичной; 100 в двоичной это 4 в десятичной, и т. д.

Если в системе счисления есть 10 цифр, её *основание* или *radix* это 10. Двоичная система имеет *основание* 2.

Важные вещи, которые полезно вспомнить: 1) *число* это число, в то время как *цифра* это термин из системы письменности, и это обычно один символ; 2) само число не меняется, когда конвертируется из одного основания в другое: меняется способ его записи (или представления в памяти).

Как сконвертировать число из одного основания в другое?

Позиционная нотация используется почти везде, это означает, что всякая цифра имеет свой вес, в зависимости от её расположения внутри числа. Если 2 расположена в самом последнем месте справа, это 2. Если она расположена в месте перед последним, это 20.

Что означает 1234?

$$10^3 \cdot 1 + 10^2 \cdot 2 + 10^1 \cdot 3 + 1 \cdot 4 = 1234 \text{ или } 1000 \cdot 1 + 100 \cdot 2 + 10 \cdot 3 + 4 = 1234$$

Та же история и для двоичных чисел, только основание там 2 вместо 10. Что означает 0b101011?

$$2^5 \cdot 1 + 2^4 \cdot 0 + 2^3 \cdot 1 + 2^2 \cdot 0 + 2^1 \cdot 1 + 2^0 \cdot 1 = 43 \text{ или } 32 \cdot 1 + 16 \cdot 0 + 8 \cdot 1 + 4 \cdot 0 + 2 \cdot 1 + 1 = 43$$

Позиционную нотацию можно противопоставить непозиционной нотации, такой как римская система записи чисел <sup>8</sup>. Вероятно, человечество перешло на позиционную нотацию, потому что так проще работать с числами (сложение, умножение, и т. д.) на бумаге, в ручную.

Действительно, двоичные числа можно складывать, вычитать, и т. д., точно также, как этому обычно обучают в школах, только доступны лишь 2 цифры.

Двоичные числа громоздки, когда их используют в исходных кодах и дампах, так что в этих случаях применяется шестнадцатеричная система. Используются цифры 0..9 и еще 6 латинских букв: A..F. Каждая шестнадцатеричная цифра

<sup>8</sup>Об эволюции способов записи чисел, см. также: [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 195-213.]

занимает 4 бита или 4 двоичных цифры, так что конвертировать из двоичной системы в шестнадцатеричную и назад, можно легко вручную, или даже в уме.

шестнадцатеричная	двоичная	десятичная
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Как понять, какое основание используется в конкретном месте?

Десятичные числа обычно записываются как есть, т.е., 1234. Но некоторые ассемблеры позволяют подчеркивать этот факт для ясности, и это число может быть дополнено суффиксом "d": 1234d.

К двоичным числам иногда спереди добавляют префикс "0b": 0b100110111 (В **GCC!**<sup>9</sup> для этого есть нестандартное расширение языка <sup>10</sup>). Есть также еще один способ: суффикс "b", например: 100110111b. В этой книге я буду пытаться придерживаться префикса "0b" для двоичных чисел.

Шестнадцатеричные числа имеют префикс "0x" в Си/Си++ и некоторых других **PL!**: 0x1234ABCD. Либо они имеют суффикс "h": 1234ABCDh — обычно так они представляются в ассемблерах и отладчиках. Если число начинается с цифры A..F, перед ним добавляется 0: 0ABCDEFh. Во времена 8-битных домашних компьютеров, был также способ записи чисел используя префикс \$, например, \$ABCD. В книге я попытаюсь придерживаться префикса "0x" для шестнадцатеричных чисел.

Нужно ли учиться конвертировать числа в уме? Таблицу шестнадцатеричных чисел из одной цифры легко запомнить. А запоминать бóльшие числа, наверное, не стоит.

Наверное, чаще всего шестнадцатеричные числа можно увидеть в **URL!**<sup>11</sup>-ах. Так кодируются буквы не из числа латинских. Например: <https://en.wiktionary.org/wiki/na%C3%AFvet%C3%A9> это **URL!** страницы в Wiktionary о слове «naïveté».

<sup>9</sup>**GCC!**

<sup>10</sup><https://gcc.gnu.org/onlinedocs/gcc/Binary-constants.html>

<sup>11</sup>**URL!**

## Восьмеричная система

Еще одна система, которая в прошлом много использовалась в программировании это восьмеричная: есть 8 цифр (0..7) и каждая описывает 3 бита, так что легко конвертировать числа туда и назад. Она почти везде была заменена шестнадцатеричной, но удивительно, в \*NIX имеется утилита использующаяся многими людьми, которая принимает на вход восьмеричное число: `chmod`.

Как знают многие пользователи \*NIX, аргумент `chmod` это число из трех цифр. Первая цифра это права владельца файла, вторая это права группы (которой файл принадлежит), третья для всех остальных. И каждая цифра может быть представлена в двоичном виде:

десятичная	двоичная	значение
7	111	<b>rwX</b>
6	110	<b>rw-</b>
5	101	<b>r-x</b>
4	100	<b>r--</b>
3	011	<b>-wX</b>
2	010	<b>-w-</b>
1	001	<b>--X</b>
0	000	<b>---</b>

Так что каждый бит привязан к флагу: `read/write/execute` (чтение/запись/исполнение).

И вот почему я вспомнил здесь о `chmod`, это потому что всё число может быть представлено как число в восьмеричной системе. Для примера возьмем 644. Когда вы запускаете `chmod 644 file`, вы выставляете права `read/write` для владельца, права `read` для группы, и снова, `read` для всех остальных. Сконвертируем число 644 из восьмеричной системы в двоичную, это будет 110100100, или (в группах по 3 бита) 110 100 100.

Теперь мы видим, что каждая тройка описывает права для владельца/группы/остальных: первая это `rw-`, вторая это `r--` и третья это `r--`.

Восьмеричная система была также популярная на старых компьютерах вроде PDP-8, потому что слово там могло содержать 12, 24 или 36 бит, и эти числа делятся на 3, так что выбор восьмеричной системы в той среде был логичен. Сейчас, все популярные компьютеры имеют размер слова/адреса 16, 32 или 64 бита, и эти числа делятся на 4, так что шестнадцатеричная система здесь удобнее.

Восьмеричная система поддерживается всеми стандартными компиляторами Си/Си++. Это иногда источник недоумения, потому что восьмеричные числа кодируются с нулем вперед, например, 0377 это 255. И иногда, вы можете сделать опечатку, и написать "09" вместо 9, и компилятор выдаст ошибку. GCC может выдать что-то вроде:

```
error: invalid digit "9" in octal constant.
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Также, восьмеричная система популярна в Java: когда IDA показывает строку с непечатаемыми символами, они кодируются в восьмеричной системе вместо шестнадцатеричной. Точно также себя ведет декомпилятор с Java JAD.

### Делимость

Когда вы видите десятичное число вроде 120, вы можете быстро понять что оно делится на 10, потому что последняя цифра это 0. Точно также, 123400 делится на 100, потому что две последних цифры это нули.

Точно также, шестнадцатеричное число 0x1230 делится на 0x10 (или 16), 0x123000 делится на 0x1000 (или 4096), и т. д.

Двоичное число 0b1000101000 делится на 0b1000 (8), и т. д.

Это свойство можно часто использовать, чтобы быстро понять, что длина какого-либо блока в памяти выровнена по некоторой границе. Например, секции в **PE!**<sup>12</sup>-файлах почти всегда начинаются с адресов заканчивающихся 3 шестнадцатеричными нулями: 0x41000, 0x10001000, и т. д. Причина в том, что почти все секции в **PE!** выровнены по границе 0x1000 (4096) байт.

### Арифметика произвольной точности и основание

Арифметика произвольной точности (multi-precision arithmetic) может использовать огромные числа, которые могут храниться в нескольких байтах. Например, ключи RSA, и открытые и закрытые, могут занимать до 4096 бит и даже больше.

В [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 265] можно найти такую идею: когда вы сохраняете число произвольной точности в нескольких байтах, всё число может быть представлено как имеющую систему счисления по основанию  $2^8 = 256$ , и каждая цифра находится в соответствующем байте. Точно также, если вы сохраняете число произвольной точности в нескольких 32-битных целочисленных значениях, каждая цифра отправляется в каждый 32-битный слот, и вы можете считать что это число записано в системе с основанием  $2^{32}$ .

### Произношение

Числа в недесятичных системах счисления обычно произносятся по одной цифре: “один-ноль-ноль-один-один-...”. Слова вроде “десять”, “тысяча”, и т. д., обычно не произносятся, потому что тогда можно спутать с десятичной системой.

### Числа с плавающей запятой

Чтобы отличать числа с плавающей запятой от целочисленных, часто, в конце добавляют “.0”, например 0.0, 123.0, и т. д.

---

<sup>12</sup>**PE!**

## 1.3. Пустая функция

Простейшая функция из всех возможных, это функция, которая ничего не делает:

Листинг 1.1: Код на Си/Си++

```
void f()
{
    return;
};
```

Скомпилируем!

### 1.3.1. x86

Для x86 и MSVC и GCC делает одинаковый код:

Листинг 1.2: Оптимизирующий GCC/MSVC (вывод на ассемблере)

```
f:
    ret
```

Тут только одна инструкция: RET, которая возвращает управление в [вызывающую ф-цию](#).

### 1.3.2. ARM

Листинг 1.3: Оптимизирующий Keil 6/2013 (Режим ARM) ASM Output

```
f      PROC
      BX      lr
      ENDP
```

Адрес возврата (**RA!**<sup>13</sup>) в ARM не сохраняется в локальном стеке, а в регистре **LR!**<sup>14</sup>. Так что инструкция BX LR делает переход по этому адресу, и это то же самое что и вернуть управление в вызывающую ф-цию.

### 1.3.3. MIPS

Есть два способа называть регистры в мире MIPS. По номеру (от \$0 до \$31) или по псевдоимени (\$V0, \$A0, и т. д.).

Вывод на ассемблере в GCC показывает регистры по номерам:

Листинг 1.4: Оптимизирующий GCC 4.4.5 (вывод на ассемблере)

```
j      $31
nop
```

<sup>13</sup>**RA!**

<sup>14</sup>**LR!**

...а **IDA!**<sup>15</sup> — по псевдоименам:

Листинг 1.5: Оптимизирующий GCC 4.4.5 (IDA)

```
j      $ra
nop
```

Первая инструкция — это инструкция перехода (J или JR), которая возвращает управление в **вызывающую ф-цию**, переходя по адресу в регистре \$31 (или \$RA).

Это аналог регистра **LR!** в ARM.

Вторая инструкция это **NOP!**<sup>16</sup>, которая ничего не делает. Пока что мы можем её игнорировать.

### Еще кое-что об именах инструкций и регистров в MIPS

Имена регистров и инструкций в мире MIPS традиционно пишутся в нижнем регистре. Но мы будем использовать верхний регистр, потому что имена инструкций и регистров других **ISA!** в этой книге так же в верхнем регистре.

#### 1.3.4. Пустые функции на практике

Не смотря на то, что пустые функции бесполезны, они довольно часто встречаются в низкоуровневом коде.

Во-первых, популярны функции, выводящие информацию в отладочный лог, например:

Листинг 1.6: Код на Си/Си++

```
void dbg_print (const char *fmt, ...)
{
#ifdef _DEBUG
    // открыть лог-файл
    // записать в лог-файл
    // закрыть лог-файл
#endif
};

void some_function()
{
    ...

    dbg_print ("we did something\n");

    ...
};
```

<sup>15</sup>**IDA!**

<sup>16</sup>**NOP!**

В не-отладочной сборке (например, “release”), `_DEBUG` не определен, так что функция `dbg_print()`, не смотря на то, что будет продолжаться вызываться в процессе исполнения, будет пустой.

Во-вторых, популярный способ защиты ПО это компиляция нескольких сборок: одной для легальных покупателей, второй — демонстрационной. Демонстрационная сборка может не иметь каких-то важных функций, например:

Листинг 1.7: Код на Си/Си++

```
void save_file ()
{
#ifdef DEMO
    // код, сохраняющий что-то
#endif
};
```

Функция `save_file()` может быть вызвана, когда пользователь кликает меню `File->Save`. Демо-версия может поставляться с отключенным пунктом меню, но даже если кракер разрешит этот пункт, будет вызываться пустая функция, в которой полезного кода нет.

IDA маркирует такие функции именами вроде `nullsub_00`, `nullsub_01`, и т. д.

## 1.4. Возврат значения

Еще одна простейшая функция это та, что возвращает некоторую константу: Вот, например:

Листинг 1.8: Код на Си/Си++

```
int f()
{
    return 123;
};
```

Скомпилируем её.

### 1.4.1. x86

И вот что делает оптимизирующий GCC:

Листинг 1.9: Оптимизирующий GCC/MSVC (вывод на ассемблере)

```
f:
    mov     eax, 123
    ret
```

Здесь только две инструкции. Первая помещает значение 123 в регистр EAX, который используется для передачи возвращаемых значений. Вторая это RET, которая возвращает управление в вызывающую функцию.

Вызывающая функция возьмет результат из регистра EAX.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

## 1.4.2. ARM

А что насчет ARM?

Листинг 1.10: Оптимизирующий Keil 6/2013 (Режим ARM) ASM Output

```
f      PROC
      MOV     r0,#0x7b ; 123
      BX     lr
      ENDP
```

ARM использует регистр R0 для возврата значений, так что здесь 123 помещается в R0.

Нужно отметить, что название инструкции MOV в x86 и ARM сбивает с толку.

На самом деле, данные не *перемещаются*, а скорее *копируются*.

## 1.4.3. MIPS

Вывод на ассемблере в GCC показывает регистры по номерам:

Листинг 1.11: Оптимизирующий GCC 4.4.5 (вывод на ассемблере)

```
j      $31
li     $2,123          # 0x7b
```

...а **IDA!**— по псевдоименам:

Листинг 1.12: Оптимизирующий GCC 4.4.5 (IDA)

```
jr     $ra
li     $v0, 0x7B
```

Так что регистр \$2 (или \$V0) используется для возврата значений. LI это “Load Immediate”, и это эквивалент MOV в MIPS.

Другая инструкция это инструкция перехода (J или JR), которая возвращает управление в [вызывающую ф-цию](#).

Но почему инструкция загрузки (LI) и инструкция перехода (J или JR) поменяны местами? Это артефакт **RISC!** и называется он “branch delay slot”.

На самом деле, нам не нужно вникать в эти детали. Нужно просто запомнить: в MIPS инструкция после инструкции перехода исполняется *перед* инструкцией перехода.

Таким образом, инструкция перехода всегда поменяна местами с той, которая должна быть исполнена перед ней.

## 1.4.4. На практике

На практике крайне часто встречаются ф-ции, которые возвращают 1 (*true*) или 0 (*false*).

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



Самые маленькие утилиты UNIX, `/bin/true` и `/bin/false` возвращают 0 и 1 соответственно, как код возврата (ноль как код возврата обычно означает успех, а ноль означает ошибку).

## 1.5. Hello, world!

Продолжим, используя знаменитый пример из книги [Брайан Керниган, Деннис Ритчи, *Язык программирования Си*, второе издание, (1988, 2009)]:

Листинг 1.13: код на Си/Си++

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

### 1.5.1. x86

#### MSVC

Компилируем в MSVC 2010:

```
cl 1.cpp /Fa1.asm
```

(Ключ `/Fa` означает сгенерировать листинг на ассемблере)

Листинг 1.14: MSVC 2010

```
CONST    SEGMENT
$SG3830 DB    'hello, world', 0AH, 00H
CONST    ENDS
PUBLIC   _main
EXTRN   _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
_main    PROC
        push    ebp
        mov     ebp, esp
        push   OFFSET $SG3830
        call   _printf
        add    esp, 4
        xor    eax, eax
        pop    ebp
        ret    0
_main    ENDP
_TEXT    ENDS
```

MSVC выдает листинги в синтаксисе Intel. Разница между синтаксисом Intel и AT&T будет рассмотрена немного позже:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Компилятор сгенерировал файл `1.obj`, который впоследствии будет слинкован линкером в `1.exe`. В нашем случае этот файл состоит из двух сегментов: `CONST` (для данных-констант) и `_TEXT` (для кода).

Строка `hello, world` в Си/Си++ имеет тип `const char[]` [Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)p176, 7.3.2], однако не имеет имени. Но компилятору нужно как-то с ней работать, поэтому он дает ей внутреннее имя `$SG3830`.

Поэтому пример можно было бы переписать вот так:

```
#include <stdio.h>

const char $SG3830[]="hello, world\n";

int main()
{
    printf($SG3830);
    return 0;
}
```

Вернемся к листингу на ассемблере. Как видно, строка заканчивается нулевым байтом — это требования стандарта Си/Си++ для строк. Больше о строках в Си/Си++: ?? (стр. ??).

В сегменте кода `_TEXT` находится пока только одна функция: `main()`. Функция `main()`, как и практически все функции, начинается с пролога и заканчивается эпилогом <sup>17</sup>.

Далее следует вызов функции `printf()`: `CALL _printf`. Перед этим вызовом адрес строки (или указатель на неё) с нашим приветствием (“Hello, world!”) при помощи инструкции `PUSH` помещается в стек.

После того, как функция `printf()` возвращает управление в функцию `main()`, адрес строки (или указатель на неё) всё ещё лежит в стеке. Так как он больше не нужен, то [указатель стека](#) (регистр `ESP`) корректируется.

`ADD ESP, 4` означает прибавить 4 к значению в регистре `ESP`.

Почему 4? Так как это 32-битный код, для передачи адреса нужно 4 байта. В x64-коде это 8 байт.

`ADD ESP, 4` эквивалентно `POP регистр`, но без использования какого-либо регистра<sup>18</sup>.

Некоторые компиляторы, например, Intel C++ Compiler, в этой же ситуации могут вместо `ADD` сгенерировать `POP ECX` (подобное можно встретить, например, в коде Oracle RDBMS, им скомпилированном), что почти то же самое, только портится значение в регистре `ECX`. Возможно, компилятор применяет `POP ECX`, потому что эта инструкция короче (1 байт у `POP` против 3 у `ADD`).

Вот пример использования `POP` вместо `ADD` из Oracle RDBMS:

<sup>17</sup>Об этом смотрите подробнее в разделе о прологе и эпилоге функции (?? (стр. ??)).

<sup>18</sup>Флаги процессора, впрочем, модифицируются

Листинг 1.15: Oracle RDBMS 10.2 Linux (файл app.o)

.text:0800029A	push	ebx
.text:0800029B	call	qksfroChild
.text:080002A0	pop	ecx

Впрочем, MSVC был замечен в подобном же.

Листинг 1.16: Minesweeper из Windows 7 32-bit

.text:0102106F	push	0
.text:01021071	call	ds:time
.text:01021077	pop	ecx

После вызова `printf()` в оригинальном коде на Си/Си++ указано `return 0` — вернуть 0 в качестве результата функции `main()`.

В сгенерированном коде это обеспечивается инструкцией `XOR EAX, EAX`.

`XOR`, как легко догадаться — «исключающее ИЛИ»<sup>19</sup>, но компиляторы часто используют его вместо простого `MOV EAX, 0` — снова потому, что опкод короче (2 байта у `XOR` против 5 у `MOV`).

Некоторые компиляторы генерируют `SUB EAX, EAX`, что значит *отнять значение в EAX от значения в EAX*, что в любом случае даст 0 в результате.

Самая последняя инструкция `RET` возвращает управление в вызывающую функцию. Обычно это код Си/Си++ **CRT!**<sup>20</sup>, который, в свою очередь, вернёт управление в **OS!**

## GCC

Теперь скомпилируем то же самое компилятором GCC 4.4.1 в Linux: `gcc 1.c -o 1`. Затем при помощи **IDA!** посмотрим как скомпилировалась функция `main()`. **IDA!**, как и MSVC, показывает код в синтаксисе Intel<sup>21</sup>.

Листинг 1.17: код в IDA!

main	proc near
var_10	= dword ptr -10h
	push ebp
	mov ebp, esp
	and esp, 0FFFFFF0h
	sub esp, 10h
	mov eax, offset aHelloWorld ; "hello, world\n"
	mov [esp+10h+var_10], eax
	call _printf

<sup>19</sup> [wikipedia](#)

<sup>20</sup> **CRT!**

<sup>21</sup> Мы также можем заставить GCC генерировать листинги в этом формате при помощи ключей `-S -masm=intel`.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

                                mov    eax, 0
                                leave
                                retn
main                             endp

```

Почти то же самое. Адрес строки `hello, world`, лежащей в сегменте данных, вначале сохраняется в `EAX`, затем записывается в стек. А ещё в прологе функции мы видим `AND ESP, 0FFFFFF0h` — эта инструкция выравнивает значение в `ESP` по 16-байтной границе, делая все значения в стеке также выровненными по этой границе (процессор более эффективно работает с переменными, расположенными в памяти по адресам кратным 4 или 16).

`SUB ESP, 10h` выделяет в стеке 16 байт. Хотя, как будет видно далее, здесь достаточно только 4.

Это происходит потому, что количество выделяемого места в локальном стеке тоже выровнено по 16-байтной границе.

Адрес строки (или указатель на строку) затем записывается прямо в стек без помощи инструкции `PUSH`. `var_10` одновременно и локальная переменная и аргумент для `printf()`. Подробнее об этом будет ниже.

Затем вызывается `printf()`.

В отличие от `MSVC`, `GCC` в компиляции без включенной оптимизации генерирует `MOV EAX, 0` вместо более короткого опкода.

Последняя инструкция `LEAVE` — это аналог команд `MOV ESP, EBP` и `POP EBP` — то есть возврат [указателя стека](#) и регистра `EBP` в первоначальное состояние. Это необходимо, т.к. в начале функции мы модифицировали регистры `ESP` и `EBP` (при помощи `MOV EBP, ESP / AND ESP, ...`).

### **GCC: Синтаксис AT&T**

Попробуем посмотреть, как выглядит то же самое в синтаксисе AT&T языка ассемблера. Этот синтаксис больше распространен в UNIX-мире.

Листинг 1.18: компилируем в GCC 4.7.3

```
gcc -S 1_1.c
```

Получим такой файл:

Листинг 1.19: GCC 4.7.3

```

.file    "1_1.c"
.section .rodata
.LC0:
.string "hello, world\n"
.text
.globl  main
.type  main, @function
main:

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.LFB0:
    .cfi_startproc
    pushl   %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl   %esp, %ebp
    .cfi_def_cfa_register 5
    andl   $-16, %esp
    subl   $16, %esp
    movl   $.LC0, (%esp)
    call   printf
    movl   $0, %eax
    leave
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
    .section      .note.GNU-stack,"",@progbits

```

Здесь много макросов (начинающихся с точки). Они нам пока не интересны.

Пока что, ради упрощения, мы можем их игнорировать (кроме макроса *.string*, при помощи которого кодируется последовательность символов, оканчивающихся нулем — такие же строки как в Си). И тогда получится следующее <sup>22</sup>:

Листинг 1.20: GCC 4.7.3

```

.LC0:
    .string "hello, world\n"
main:
    pushl   %ebp
    movl   %esp, %ebp
    andl   $-16, %esp
    subl   $16, %esp
    movl   $.LC0, (%esp)
    call   printf
    movl   $0, %eax
    leave
    ret

```

Основные отличия синтаксиса Intel и AT&T следующие:

- Операнды записываются наоборот.

В Intel-синтаксисе:

<инструкция> <операнд назначения> <операнд-источник>.

В AT&T-синтаксисе:

<инструкция> <операнд-источник> <операнд назначения>.

<sup>22</sup>Кстати, для уменьшения генерации «лишних» макросов, можно использовать такой ключ GCC: *-fno-asynchronous-unwind-tables*

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Чтобы легче понимать разницу, можно запомнить следующее: когда вы работаете с синтаксисом Intel — можете в уме ставить знак равенства (=) между операндами, а когда с синтаксисом AT&T — мысленно ставьте стрелку направо (→) <sup>23</sup>.

- AT&T: Перед именами регистров ставится символ процента (%), а перед числами символ доллара (\$). Вместо квадратных скобок используются круглые.
- AT&T: К каждой инструкции добавляется специальный символ, определяющий тип данных:
  - q — quad (64 бита)
  - l — long (32 бита)
  - w — word (16 бит)
  - b — byte (8 бит)

Возвращаясь к результату компиляции: он идентичен тому, который мы посмотрели в **IDA!**. Одна мелочь: 0FFFFFFF0h записывается как \$-16. Это то же самое: 16 в десятичной системе это 0x10 в шестнадцатеричной. -0x10 будет как раз 0xFFFFFFFF0 (в рамках 32-битных чисел).

Возвращаемый результат устанавливается в 0 обычной инструкцией MOV, а не XOR. MOV просто загружает значение в регистр. Её название не очень удачное (данные не перемещаются, а копируются). В других архитектурах подобная инструкция обычно носит название «LOAD» или «STORE» или что-то в этом роде.

### Коррекция (патчинг) строки (Win32)

Мы можем легко найти строку “hello, world” в исполняемом файле при помощи Hiew:

```

Hiew: hw_spanish.exe
C:\tmp\hw_spanish.exe  FWO ----- PE+.00000001`40003000 Hiew 8.02
.400025E0: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.400025F0: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.40003000: 68 65 6C 6C-6F 2C 20 77-6F 72 6C 64-0A 00 00 00  hello, world
.40003010: 01 00 00 00-FE FF FF FF-FF FF FF FF-00 00 00 00
.40003020: 32 A2 DF 2D-99 2B 00 00-CD 5D 20 D2-66 D4 FF FF  2ö-Ö+ =] ПfL
.40003030: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.40003040: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
  
```

Рис. 1.1: Hiew

Можем перевести наше сообщение на испанский язык:

<sup>23</sup>Кстати, в некоторых стандартных функциях библиотеки Си (например, memcpy(), strcpy()) так же применяется расстановка аргументов как в синтаксисе Intel: вначале указатель в памяти на блок назначения, затем указатель на блок-источник.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

Hiew: hw_spanish.exe
C:\tmp\hw_spanish.exe  FWO EDITMODE  PE+ 00000000`0000120D Hiew 8.02
000011E0: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
000011F0: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
00001200: 68 6F 6C 61-2C 20 6D 75-6E 64 6F 0A-00 00 00 00 hola, mundo
00001210: 01 00 00 00-FE FF FF FF-FF FF FF FF-00 00 00 00
00001220: 32 A2 DF 2D-99 2B 00 00-CD 5D 20 D2-66 D4 FF FF 2ó-Ö+ =] Tfl
00001230: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
00001240: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00

```

Рис. 1.2: Hiew

Испанский текст на 1 байт короче английского, так что добавляем в конце байт 0x0A (\n) и нулевой байт.

Работает.

Что если мы хотим вставить более длинное сообщение? После оригинального текста на английском есть какие-то нулевые байты. Трудно сказать, можно ли их перезаписывать: они могут где-то использоваться в **CRT!**-коде, а может и нет. Так или иначе, вы можете их перезаписывать, только если вы действительно знаете, что делаете.

### Коррекция строки (Linux x64)

Попробуем пропатчить исполняемый файл для Linux x64 используя rada.re:

Листинг 1.21: Сессия в rada.re

```

dennis@bigbox ~/tmp % gcc hw.c

dennis@bigbox ~/tmp % radare2 a.out
-- SHALL WE PLAY A GAME?
[0x00400430]> / hello
Searching 5 bytes from 0x00400000 to 0x00601040: 68 65 6c 6c 6f
Searching 5 bytes in [0x400000-0x601040]
hits: 1
0x004005c4 hit0_0 .HHhello, world;0.

[0x00400430]> s 0x004005c4

[0x004005c4]> px
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x004005c4 6865 6c6c 6f2c 2077 6f72 6c64 0000 0000 hello, world....
0x004005d4 011b 033b 3000 0000 0500 0000 1cfe ffff ...;0.....
0x004005e4 7c00 0000 5cfe ffff 4c00 0000 52ff ffff |...\...L...R...
0x004005f4 a400 0000 6cff ffff c400 0000 dcff ffff ....l.....
0x00400604 0c01 0000 1400 0000 0000 0000 017a 5200 .....zR.
0x00400614 0178 1001 1b0c 0708 9001 0710 1400 0000 .x.....
0x00400624 1c00 0000 08fe ffff 2a00 0000 0000 0000 .....*.....
0x00400634 0000 0000 1400 0000 0000 0000 017a 5200 .....zR.

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

0x00400644 0178 1001 1b0c 0708 9001 0000 2400 0000 .x.....$....
0x00400654 1c00 0000 98fd ffff 3000 0000 000e 1046 .....0.....F
0x00400664 0e18 4a0f 0b77 0880 003f 1a3b 2a33 2422 ..J..w...?;*3$"
0x00400674 0000 0000 1c00 0000 4400 0000 a6fe ffff .....D.....
0x00400684 1500 0000 0041 0e10 8602 430d 0650 0c07 .....A....C..P..
0x00400694 0800 0000 4400 0000 6400 0000 a0fe ffff ....D...d.....
0x004006a4 6500 0000 0042 0e10 8f02 420e 188e 0345 e....B....B....E
0x004006b4 0e20 8d04 420e 288c 0548 0e30 8606 480e . . .B.(.H.0..H.

```

```

[0x004005c4]> oo+
File a.out reopened in read-write mode

```

```

[0x004005c4]> w hola, mundo\x00

```

```

[0x004005c4]> q

```

```

dennis@bigbox ~/tmp % ./a.out
hola, mundo

```

Что я здесь делаю: ищу строку «hello» используя команду `/`, я затем я выставляю курсор (*seek* в терминах `gdb`) на этот адрес. Потом я хочу удостовериться, что это действительно нужное место: `rx` выводит байты по этому адресу. `oo+` переключает `gdb` в режим *чтения-записи*. `w` записывает ASCII-строку на месте курсора (*seek*). Нужно отметить `\x00` в конце — это нулевой байт. `q` заканчивает работу.

### Это реальная история взлома ПО

Некое ПО обрабатывало изображения, и когда не было зарегистрировано, оно добавляло водяные знаки, вроде “This image was processed by evaluation version of [software name]”, поперек картинки. Мы попробовали от балды: нашли эту строку в исполняемом файле и забили пробелами. Водяные знаки пропали. Технически, они продолжали добавляться. При помощи соответствующих функций Qt, надпись продолжала добавляться в итоговое изображение. Но добавление пробелов не меняло само изображение...

### Локализация ПО во времена MS-DOS

Описанный способ был очень распространен для перевода ПО под MS-DOS на русский язык в 1980-е и 1990-е. Эта техника доступна даже для тех, кто вообще не разбирается в машинном коде и форматах исполняемых файлов. Новая строка не должна быть длиннее старой, потому что имеется риск затереть какую-то другую переменную или код. Русские слова и предложения обычно немного длиннее английских, так что *локализованное* ПО содержало массу странных акронимов и труднопонятных сокращений.



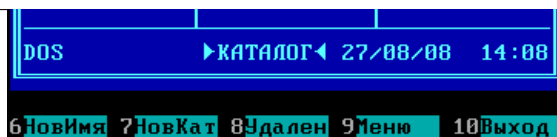


Рис. 1.3: Русифицированный Norton Commander 5.51

Вероятно, так было и с другими языками в других странах.

В строках в Delphi, длина строки также должна быть поправлена, если нужно.

## 1.5.2. x86-64

### MSVC: x86-64

Попробуем также 64-битный MSVC:

Листинг 1.22: MSVC 2012 x64

```
$SG2989 DB      'hello, world', 0AH, 00H

main PROC
    sub     rsp, 40
    lea    rcx, OFFSET FLAT:$SG2989
    call   printf
    xor    eax, eax
    add    rsp, 40
    ret    0
main     ENDP
```

В x86-64 все регистры были расширены до 64-х бит и теперь имеют префикс R-. Чтобы поменьше задействовать стек (иными словами, поменьше обращаться к кэшу и внешней памяти), уже давно имелся довольно популярный метод передачи аргументов функции через регистры (*fastcall*) ?? (стр. ??). Т.е. часть аргументов функции передается через регистры и часть — через стек. В Win64 первые 4 аргумента функции передаются через регистры RCX, RDX, R8, R9. Это мы здесь и видим: указатель на строку в `printf()` теперь передается не через стек, а через регистр RCX. Указатели теперь 64-битные, так что они передаются через 64-битные части регистров (имеющие префикс R-). Но для обратной совместимости можно обращаться и к нижним 32 битам регистров используя префикс E-. Вот как выглядит регистр RAX/EAX/AX/AL в x86-64:

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
RAX <sup>x64</sup>							
						EAX	
						AX	
						AH	AL

Функция `main()` возвращает значение типа *int*, который в Си/Си++, надо полагать, для лучшей совместимости и переносимости, оставили 32-битным. Вот

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

почему в конце функции `main()` обнуляется не `RAX`, а `EAX`, т.е. 32-битная часть регистра. Также видно, что 40 байт выделяются в локальном стеке. Это «shadow space» которое мы будем рассматривать позже: ?? (стр. ??).

## GCC: x86-64

Попробуем GCC в 64-битном Linux:

Листинг 1.23: GCC 4.4.6 x64

```
.string "hello, world\n"
main:
    sub    rsp, 8
    mov    edi, OFFSET FLAT:.LC0 ; "hello, world\n"
    xor    eax, eax ; количество переданных векторных регистров
    call   printf
    xor    eax, eax
    add    rsp, 8
    ret
```

В Linux, \*BSD и Mac OS X для x86-64 также принят способ передачи аргументов функции через регистры [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)]<sup>24</sup>.

6 первых аргументов передаются через регистры `RDI`, `RSI`, `RDY`, `RCX`, `R8`, `R9`, а остальные — через стек.

Так что указатель на строку передается через `EDI` (32-битную часть регистра). Но почему не через 64-битную часть, `RDI`?

Важно запомнить, что в 64-битном режиме все инструкции `MOV`, записывающие что-либо в младшую 32-битную часть регистра, обнуляют старшие 32-бита (это можно найти в документации от Intel: ?? (стр. ??)). То есть, инструкция `MOV EAX, 011223344h` корректно запишет это значение в `RAX`, старшие биты сбросятся в ноль.

Если посмотреть в **IDA!** скомпилированный объектный файл (`.o`), увидим также опкоды всех инструкций<sup>25</sup>:

Листинг 1.24: GCC 4.4.6 x64

```
.text:00000000004004D0      main  proc near
.text:00000000004004D0 48 83 EC 08      sub    rsp, 8
.text:00000000004004D4 BF E8 05 40 00   mov    edi, offset format ; "hello,
world\n"
.text:00000000004004D9 31 C0           xor    eax, eax
.text:00000000004004DB E8 D8 FE FF FF   call   _printf
.text:00000000004004E0 31 C0           xor    eax, eax
.text:00000000004004E2 48 83 C4 08     add    rsp, 8
.text:00000000004004E6 C3             retn
.text:00000000004004E6      main  endp
```

<sup>24</sup>Также доступно здесь: <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>

<sup>25</sup>Это нужно задать в **Options** → **Disassembly** → **Number of opcode bytes**

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Как видно, инструкция, записывающая в EDI по адресу 0x4004D4, занимает 5 байт. Та же инструкция, записывающая 64-битное значение в RDI, занимает 7 байт. Возможно, GCC решил немного сэкономить. К тому же, вероятно, он уверен, что сегмент данных, где хранится строка, никогда не будет расположен в адресах выше 4GiB.

Здесь мы также видим обнуление регистра EAX перед вызовом `printf()`. Это делается потому что по упомянутому выше стандарту передачи аргументов в \*NIX для x86-64 в EAX передается количество задействованных векторных регистров.

### Коррекция (патчинг) адреса (Win64)

Если наш пример скомпилирован в MSVC 2013 используя опцию `/MD` (подразумеваемая меньший исполняемый файл из-за внешнего связывания файла `MSVCR*.DLL`), функция `main()` идет первой, и её легко найти:

```

Hiew: hw2.exe
C:\tmp\hw2.exe @FWO EDITMODE a64 PE+ 00000000`00000404 Hiew 8.02 (c)SEN
00000400: 4883EC28 sub rsp,028 ; '('
00000404: 488D0DF51F0000 lea rcx,[000002400]
0000040B: FF15D7100000 call q,[0000014E8]
00000411: 33C0 xor eax,eax
00000413: 4883C428 add rsp,028 ; '('
00000417: C3 retq
00000418: 4883EC28 sub rsp,028 ; '('
0000041C: B84D5A0000 mov eax,000005A4D ; ' ZM'
00000421: 663905D8EFFFFFF cmp [-00000C00],ax
00000428: 7404 jz 00000042E
0000043F: 813850450000 cmp d,[rax],000004550 ; ' EP'
00000445: 75E3 jnz 00000042A
00000447: B90B020000 mov ecx,00000020B
0000044C: 66394818 cmp [rax][018],cx
00000450: 75D8 jnz 00000042A
00000452: 33C9 xor ecx,ecx
00000454: 83B8840000000E cmp d,[rax][000000084],00E
0000045B: 7609 jbe 000000466
0000045D: 3988F8000000 cmp [rax][0000000F8],ecx

```

Рис. 1.4: Hiew

В качестве эксперимента, мы можем [инкрементировать](#) адрес на 1:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

C:\tmp\hw2.exe  FUO  -----  a64  PE+.00000001`4000100B  Hiew 8.02 (c)SEN
.40001000: 4883EC28      sub     rsp,028 ; '('
.40001004: 488D0DF61F0000  lea    rcx,[00000001`40003001] ;'ello, w
.4000100B: FF15D7100000    call   printf
.40001011: 33C0        xor     eax,eax
.40001013: 4883C428     add     rsp,028 ; '('
.40001017: C3          retn   ; ^^^^
.40001018: 4883EC28     sub     rsp,028 ; '('
.4000101C: B84D5A0000   mov     eax,00005A4D ;' ZM'
.40001021: 663905D8EFFFFF cmp    [00000001`40000000],ax
.40001028: 7404        jz     .00000001`4000102E --B2
.4000102A: 33C9        xor     ecx,ecx
.4000102C: EB38        jmps   .00000001`40001066 --B3
.4000102E: 48630507F0FFFF movsxd rax,d,[00000001`4000003C] --B4
.40001035: 488D0DC4EFFFFF lea    rcx,[00000001`40000000]
.4000103C: 4803C1      add     rax,rcx
.4000103F: 813850450000  cmp    d,[rax],000004550 ;' EP'
.40001045: 75E3        jnz   .00000001`4000102A --B5
.40001047: B90B020000   mov     ecx,0000020B
.4000104C: 66394818     cmp    [rax][0x18],cx
.40001050: 75D8        jnz   .00000001`4000102A --B5
.40001052: 33C9        xor     ecx,ecx
.40001054: 83B884000000E cmp    d,[rax][00000084],00E
.4000105B: 7609        jbe   .00000001`40001066 --B3
.4000105D: 3988F8000000  cmp    [rax][000000F8],ecx
1Help 2PutBk 3Edit 4Mode 5Goto 6Refer 7Search 8Header 9Files 10Quit 11Hem

```

Рис. 1.5: Hiew

Hiew показывает строку «ello, world». И когда мы запускаем исполняемый файл, именно эта строка и выводится.

### Выбор другой строки из исполняемого файла (Linux x64)

Исполняемый файл, если скомпилировать используя GCC 5.4.0 на Linux x64, имеет множество других строк: в основном, это имена импортированных функций и имена библиотек.

Запускаю objdump, чтобы посмотреть содержимое всех секций скомпилированного файла:

```

% objdump -s a.out

a.out:      file format elf64-x86-64

Contents of section .interp:
 400238 2f6c6962 36342f6c 642d6c69 6e75782d /lib64/ld-linux-
 400248 7838362d 36342e73 6f2e3200          x86-64.so.2.
Contents of section .note.ABI-tag:
 400254 04000000 10000000 01000000 474e5500 .....GNU.
 400264 00000000 02000000 06000000 20000000 .....
Contents of section .note.gnu.build-id:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
400274 04000000 14000000 03000000 474e5500 .....GNU.
400284 fe461178 5bb710b4 bbf2aca8 5ec1ec10 .F.x[.....^...
400294 cf3f7ae4                               .?z.
```

```
...
```

Не проблема передать адрес текстовой строки «/lib64/ld-linux-x86-64.so.2» в вызов `printf()`:

```
#include <stdio.h>

int main()
{
    printf(0x400238);
    return 0;
}
```

Трудно поверить, но этот код печатает вышеуказанную строку.

Измените адрес на `0x400260`, и напечатается строка «GNU». Адрес точен для конкретной версии GCC, GNU toolset, и т. д. На вашей системе, исполняемый файл может быть немного другой, и все адреса тоже будут другими. Также, добавление/удаление кода из исходных кодов, скорее всего, сдвинет все адреса вперед или назад.

### 1.5.3. ARM

Для экспериментов с процессором ARM было использовано несколько компиляторов:

- Популярный в embedded-среде Keil Release 6/2013.
- Apple Xcode 4.6.3 с компилятором LLVM-GCC 4.2 <sup>26</sup>.
- GCC 4.9 (Linaro) (для ARM64), доступный в виде исполняемого файла для win32 на <http://www.linaro.org/projects/armv8/>.

Везде в этой книге, если не указано иное, идет речь о 32-битном ARM (включая режимы Thumb и Thumb-2). Когда речь идет о 64-битном ARM, он называется здесь ARM64.

#### Неоптимизирующий Keil 6/2013 (Режим ARM)

Для начала скомпилируем наш пример в Keil:

```
armcc.exe --arm --c90 -00 1.c
```

Компилятор *armcc* генерирует листинг на ассемблере в формате Intel. Этот листинг содержит некоторые высокоуровневые макросы, связанные с ARM <sup>27</sup>, а

<sup>26</sup>Это действительно так: Apple Xcode 4.6.3 использует open-сорсный GCC как компилятор переднего плана и кодогенератор LLVM

<sup>27</sup>например, он показывает инструкции PUSH/POP, отсутствующие в режиме ARM

нам важнее увидеть инструкции «как есть», так что посмотрим скомпилированный результат в **IDA!**.

Листинг 1.25: Неоптимизирующий Keil 6/2013 (Режим ARM) **IDA!**

```
.text:00000000          main
.text:00000000 10 40 2D E9      STMFD   SP!, {R4,LR}
.text:00000004 1E 0E 8F E2      ADR     R0, aHelloWorld ; "hello, world"
.text:00000008 15 19 00 EB      BL      __2printf
.text:0000000C 00 00 A0 E3      MOV     R0, #0
.text:00000010 10 80 BD E8      LDMFD  SP!, {R4,PC}

.text:000001EC 68 65 6C 6C+aHelloWorld DCB "hello, world",0 ; DATA XREF:
main+4
```

В вышеприведённом примере можно легко увидеть, что каждая инструкция имеет размер 4 байта. Действительно, ведь мы же компилировали наш код для режима ARM, а не Thumb.

Самая первая инструкция, **STMFD SP!, {R4,LR}**<sup>28</sup>, работает как инструкция **PUSH** в x86, записывая значения двух регистров (R4 и **LR!**) в стек. Действительно, в выдаваемом листинге на ассемблере компилятор *armcc* для упрощения указывает здесь инструкцию **PUSH {r4,lr}**. Но это не совсем точно, инструкция **PUSH** доступна только в режиме Thumb, поэтому, во избежание путаницы, я предложил работать в **IDA!**.

Итак, эта инструкция уменьшает **SP!**<sup>30</sup>, чтобы он указывал на место в стеке, свободное для записи новых значений, затем записывает значения регистров R4 и **LR!** по адресу в памяти, на который указывает изменённый регистр **SP!**.

Эта инструкция, как и инструкция **PUSH** в режиме Thumb, может сохранить в стеке одновременно несколько значений регистров, что может быть очень удобно. Кстати, такого в x86 нет. Также следует заметить, что **STMFD** — генерализация инструкции **PUSH** (то есть расширяет её возможности), потому что может работать с любым регистром, а не только с **SP!**. Другими словами, **STMFD** можно использовать для записи набора регистров в указанном месте памяти.

Инструкция **ADR R0, aHelloWorld** прибавляет или отнимает значение регистра **PC!**<sup>31</sup> к смещению, где хранится строка `hello, world`. Причем здесь **PC!**, можно спросить? Притом, что это так называемый «адресно-независимый код»<sup>32</sup>. Он предназначен для исполнения будучи не привязанным к каким-либо адресам в памяти. Другими словами, это относительная от **PC!** адресация. В опкоде инструкции **ADR** указывается разница между адресом этой инструкции и местом, где хранится строка. Эта разница всегда будет постоянной, вне зависимости от того, куда был загружен **OS!** наш код. Поэтому всё, что нужно — это прибавить адрес текущей инструкции (из **PC!**), чтобы получить текущий абсолютный адрес нашей Си-строки.

<sup>28</sup>**STMFD!**<sup>29</sup>

<sup>30</sup>**SP!**

<sup>31</sup>**PC!**

<sup>32</sup>читайте больше об этом в соответствующем разделе (?? (стр. ??))

Инструкция `BL __2printf`<sup>33</sup> вызывает функцию `printf()`. Работа этой инструкции состоит из двух фаз:

- записать адрес после инструкции `BL (0xC)` в регистр **LR!**;
- передать управление в `printf()`, записав адрес этой функции в регистр **PC!**.

Ведь когда функция `printf()` закончит работу, нужно знать, куда вернуть управление, поэтому закончив работу, всякая функция передает управление по адресу, записанному в регистре **LR!**.

В этом разница между «чистыми» **RISC!**-процессорами вроде ARM и **CISC!**<sup>34</sup>-процессорами как x86, где адрес возврата обычно записывается в стек (**??** (стр. **??**)).

Кстати, 32-битный абсолютный адрес (либо смещение) невозможно закодировать в 32-битной инструкции `BL`, в ней есть место только для 24-х бит. Поскольку все инструкции в режиме ARM имеют длину 4 байта (32 бита) и инструкции могут находиться только по адресам кратным 4, то последние 2 бита (всегда нулевых) можно не кодировать. В итоге имеем 26 бит, при помощи которых можно закодировать  $current\_PC \pm \approx 32M$ .

Следующая инструкция `MOV R0, #0`<sup>35</sup> просто записывает 0 в регистр R0. Ведь наша Си-функция возвращает 0, а возвращаемое значение всякая функция оставляет в R0.

Последняя инструкция `LDMFD SP!, R4, PC`<sup>36</sup>. Она загружает из стека (или любого другого места в памяти) значения для сохранения их в R4 и **PC!**, увеличивая **указатель стека SP!**. Здесь она работает как аналог `POP`.

N.B. Самая первая инструкция `STMFD` сохранила в стеке R4 и **LR!**, а *восстанавливаются* во время исполнения `LDMFD` регистры R4 и **PC!**.

Как мы уже знаем, в регистре **LR!** обычно сохраняется адрес места, куда нужно всякой функции вернуть управление. Самая первая инструкция сохраняет это значение в стеке, потому что наша функция `main()` позже будет сама пользоваться этим регистром в момент вызова `printf()`. А затем, в конце функции, это значение можно сразу записать прямо в **PC!**, таким образом, передав управление туда, откуда была вызвана наша функция.

Так как функция `main()` обычно самая главная в Си/Си++, управление будет возвращено в загрузчик **OS!**, либо куда-то в **CRT!** или что-то в этом роде.

Всё это позволяет избавиться от инструкции `BX LR` в самом конце функции.

`DCB` — директива ассемблера, описывающая массивы байт или ASCII-строк, аналог директивы `DB` в x86-ассемблере.

<sup>33</sup>Branch with Link

<sup>34</sup>**CISC!**

<sup>35</sup>Означает MOVE

<sup>36</sup>**LDMFD!**<sup>37</sup> — это инструкция, обратная **STMFD!**

## Неоптимизирующий Keil 6/2013 (Режим Thumb)

Скомпилируем тот же пример в Keil для режима Thumb:

```
armcc.exe --thumb --c90 -00 1.c
```

Получим (в **IDA!**):

Листинг 1.26: Неоптимизирующий Keil 6/2013 (Режим Thumb) + **IDA!**

```
.text:00000000          main
.text:00000000 10 B5          PUSH    {R4,LR}
.text:00000002 C0 A0          ADR     R0, aHelloWorld ; "hello, world"
.text:00000004 06 F0 2E F9    BL      __2printf
.text:00000008 00 20          MOVS   R0, #0
.text:0000000A 10 BD          POP    {R4,PC}

.text:00000304 68 65 6C 6C+aHelloWorld DCB "hello, world",0 ; DATA XREF:
main+2
```

Сразу бросаются в глаза двухбайтные (16-битные) опкоды — это, как уже было отмечено, Thumb.

Кроме инструкции BL. Но на самом деле она состоит из двух 16-битных инструкций. Это потому что в одном 16-битном опкоде слишком мало места для задания смещения, по которому находится функция printf(). Так что первая 16-битная инструкция загружает старшие 10 бит смещения, а вторая — младшие 11 бит смещения.

Как уже было упомянуто, все инструкции в Thumb-режиме имеют длину 2 байта (или 16 бит). Поэтому невозможна такая ситуация, когда Thumb-инструкция начинается по нечетному адресу.

Учитывая сказанное, последний бит адреса можно не кодировать. Таким образом, в Thumb-инструкции BL можно закодировать адрес  $current\_PC \pm \approx 2M$ .

Остальные инструкции в функции (PUSH и POP) здесь работают почти так же, как и описанные STMFD/LDMFD, только регистр **SP!** здесь не указывается явно. ADR работает так же, как и в предыдущем примере. MOVS записывает 0 в регистр R0 для возврата нуля.

## Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

Xcode 4.6.3 без включенной оптимизации выдает слишком много лишнего кода, поэтому включим оптимизацию компилятора (ключ -O3), потому что там меньше инструкций.

Листинг 1.27: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

```
__text:000028C4          _hello_world
__text:000028C4 80 40 2D E9    STMFD   SP!, {R7,LR}
__text:000028C8 86 06 01 E3    MOV     R0, #0x1686
__text:000028CC 0D 70 A0 E1    MOV     R7, SP
__text:000028D0 00 00 40 E3    MOVT   R0, #0
__text:000028D4 00 00 8F E0    ADD    R0, PC, R0
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



__text:000028D8	C3 05 00 EB	BL	_puts
__text:000028DC	00 00 A0 E3	MOV	R0, #0
__text:000028E0	80 80 BD E8	LDMFD	SP!, {R7,PC}
__cstring:00003F62	48 65 6C 6C	+aHelloWorld_0	DCB "Hello world!",0

Инструкции STMFD и LDMFD нам уже знакомы.

Инструкция MOV просто записывает число 0x1686 в регистр R0 — это смещение, указывающее на строку «Hello world!».

Регистр R7 (по стандарту, принятому в [iOS ABI Function Call Guide, (2010)]<sup>38</sup>) это frame pointer, о нем будет рассказано позже.

Инструкция MOVT R0, #0 (MOVE Top) записывает 0 в старшие 16 бит регистра. Дело в том, что обычная инструкция MOV в режиме ARM может записывать какое-либо значение только в младшие 16 бит регистра, ведь в ней нельзя закодировать больше. Помните, что в режиме ARM опкоды всех инструкций ограничены длиной в 32 бита. Конечно, это ограничение не касается перемещений данных между регистрами.

Поэтому для записи в старшие биты (с 16-го по 31-й включительно) существует дополнительная команда MOVT. Впрочем, здесь её использование избыточно, потому что инструкция MOV R0, #0x1686 выше и так обнулила старшую часть регистра. Возможно, это недочет компилятора.

Инструкция ADD R0, PC, R0 прибавляет **PC!** к R0 для вычисления действительного адреса строки «Hello world!». Как нам уже известно, это «адресно-независимый код», поэтому такая корректура необходима.

Инструкция BL вызывает puts() вместо printf().

LLVM заменил вызов printf() на puts(). Действительно, printf() с одним аргументом это почти аналог puts().

Почти, если принять условие, что в строке не будет управляющих символов printf(), начинающихся со знака процента. Тогда эффект от работы этих двух функций будет разным <sup>39</sup>.

Зачем компилятор заменил один вызов на другой? Наверное потому что puts() работает быстрее <sup>40</sup>. Видимо потому что puts() проталкивает символы в stdout не сравнивая каждый со знаком процента.

Далее уже знакомая инструкция MOV R0, #0, служащая для установки в 0 возвращаемого значения функции.

<sup>38</sup>Также доступно здесь: <http://developer.apple.com/library/ios/documentation/Xcode/Conceptual/iPhoneOSABIReference/iPhoneOSABIReference.pdf>

<sup>39</sup>Также нужно заметить, что puts() не требует символа перевода строки '\n' в конце строки, поэтому его здесь нет.

<sup>40</sup>[cislant.de/projects/gcc\\_printf/gcc\\_printf.html](http://cislant.de/projects/gcc_printf/gcc_printf.html)

### Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

По умолчанию Xcode 4.6.3 генерирует код для режима Thumb-2 примерно в такой манере:

Листинг 1.28: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

```

__text:00002B6C          _hello_world
__text:00002B6C 80 B5          PUSH          {R7,LR}
__text:00002B6E 41 F2 D8 30    MOVW         R0, #0x13D8
__text:00002B72 6F 46          MOV          R7, SP
__text:00002B74 C0 F2 00 00    MOVT.W      R0, #0
__text:00002B78 78 44          ADD         R0, PC
__text:00002B7A 01 F0 38 EA    BLX         _puts
__text:00002B7E 00 20          MOVS        R0, #0
__text:00002B80 80 BD          POP         {R7,PC}

...

__cstring:00003E70 48 65 6C 6C 6F 20+aHelloWorld DCB "Hello world!",0xA,0

```

Инструкции BL и BLX в Thumb, как мы помним, кодируются как пара 16-битных инструкций, а в Thumb-2 эти *суррогатные* опкоды расширены так, что новые инструкции кодируются здесь как 32-битные инструкции. Это можно заметить по тому что опкоды Thumb-2 инструкций всегда начинаются с 0xFx либо с 0xEx. Но в листинге **IDA!** байты опкода переставлены местами. Это из-за того, что в процессоре ARM инструкции кодируются так: в начале последний байт, потом первый (для Thumb и Thumb-2 режима), либо, (для инструкций в режиме ARM) в начале четвертый байт, затем третий, второй и первый (т.е. другой *endianness*).

Вот так байты следуют в листингах IDA:

- для режимов ARM и ARM64: 4-3-2-1;
- для режима Thumb: 2-1;
- для пары 16-битных инструкций в режиме Thumb-2: 2-1-4-3.

Так что мы видим здесь что инструкции MOVW, MOVT.W и BLX начинаются с 0xFx.

Одна из Thumb-2 инструкций это MOVW R0, #0x13D8 — она записывает 16-битное число в младшую часть регистра R0, очищая старшие биты.

Ещё MOVT.W R0, #0 — эта инструкция работает так же, как и MOVT из предыдущего примера, но она работает в Thumb-2.

Помимо прочих отличий, здесь используется инструкция BLX вместо BL. Отличие в том, что помимо сохранения адреса возврата в регистре **LR!** и передаче управления в функцию puts(), происходит смена режима процессора с Thumb/Thumb-2 на режим ARM (либо назад). Здесь это нужно потому, что инструкция, куда ведет переход, выглядит так (она закодирована в режиме ARM):

```

__symbolstub1:00003FEC _puts          ; CODE XREF: _hello_world+E
__symbolstub1:00003FEC 44 F0 9F E5    LDR PC, =__imp__puts

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Это просто переход на место, где записан адрес `puts()` в секции импортов. И так, внимательный читатель может задать справедливый вопрос: почему бы не вызывать `puts()` сразу в том же месте кода, где он нужен? Но это не очень выгодно из-за экономии места и вот почему.

Практически любая программа использует внешние динамические библиотеки (будь то DLL в Windows, `.so` в \*NIX либо `.dylib` в Mac OS X). В динамических библиотеках находятся часто используемые библиотечные функции, в том числе стандартная функция Си `puts()`.

В исполняемом бинарном файле (Windows PE `.exe`, ELF либо Mach-O) имеется секция импортов, список символов (функций либо глобальных переменных) импортируемых из внешних модулей, а также названия самих модулей. Загрузчик **OS!** загружает необходимые модули и, перебирая импортируемые символы в основном модуле, проставляет правильные адреса каждого символа. В нашем случае, `__imp_puts` это 32-битная переменная, куда загрузчик **OS!** запишет правильный адрес этой же функции во внешней библиотеке. Так что инструкция LDR просто берет 32-битное значение из этой переменной, и, записывая его в регистр **PC!**, просто передает туда управление. Чтобы уменьшить время работы загрузчика **OS!**, нужно чтобы ему пришлось записать адрес каждого символа только один раз, в соответствующее, выделенное для них, место.

К тому же, как мы уже убедились, нельзя одной инструкцией загрузить в регистр 32-битное число без обращений к памяти. Так что наиболее оптимально выделить отдельную функцию, работающую в режиме ARM, чья единственная цель — передавать управление дальше, в динамическую библиотеку. И затем ссылаться на эту короткую функцию из одной инструкции (так называемую [thunk-функцию](#)) из Thumb-кода.

Кстати, в предыдущем примере (скомпилированном для режима ARM), переход при помощи инструкции BL ведет на такую же [thunk-функцию](#), однако режим процессора не переключается (отсюда отсутствие «X» в мнемонике инструкции).

## Еще о thunk-функциях

Thunk-функции трудновато понять, скорее всего, из-за путаницы в терминах. Проще всего представлять их как адаптеры-переходники из одного типа разъемов в другой. Например, адаптер, позволяющий вставить в американскую розетку британскую вилку, или наоборот. Thunk-функции также иногда называются *wrapper-ами*. *Wrap* в английском языке это *обертывать, завертывать*. Вот еще несколько описаний этих функций:

“A piece of coding which provides an address:”, according to P. Z. Ingerman, who invented thunks in 1961 as a way of binding actual parameters to their formal definitions in Algol-60 procedure calls. If a procedure is called with an expression in the place of a formal parameter, the compiler generates a thunk which computes

the expression and leaves the address of the result in some standard location.

...

Microsoft and IBM have both defined, in their Intel-based systems, a “16-bit environment” (with bletcherous segment registers and 64K address limits) and a “32-bit environment” (with flat addressing and semi-real memory management). The two environments can both be running on the same computer and OS (thanks to what is called, in the Microsoft world, WOW which stands for Windows On Windows). MS and IBM have both decided that the process of getting from 16- to 32-bit and vice versa is called a “thunk”; for Windows 95, there is even a tool, THUNK.EXE, called a “thunk compiler”.

( [The Jargon File](#) )

Еще один пример мы можем найти в библиотеке LAPACK — (“Linear Algebra PACKage”) написанная на FORTRAN. Разработчики на Си/Си++ также хотят использовать LAPACK, но переписывать её на Си/Си++, а затем поддерживать несколько версий, это безумие. Так что имеются короткие функции на Си вызываемые из Си/Си++-среды, которые, в свою очередь, вызывают функции на FORTRAN, и почти ничего больше не делают:

```
double Blas_Dot_Prod(const LaVectorDouble &dx, const LaVectorDouble &dy)
{
    assert(dx.size()==dy.size());
    integer n = dx.size();
    integer incx = dx.inc(), incy = dy.inc();

    return F77NAME(ddot>(&n, &dx(0), &incx, &dy(0), &incy);
}
```

Такие ф-ции еще называют “wrappers” (т.е., “обертка”).

## ARM64

### GCC

Компилируем пример в GCC 4.8.1 для ARM64:

Листинг 1.29: Неоптимизирующий GCC 4.8.1 + objdump

```
1 000000000400590 <main>:
2 400590: a9bf7bfd stp x29, x30, [sp,#-16]!
3 400594: 910003fd mov x29, sp
4 400598: 90000000 adrp x0, 400000 <_init-0x3b8>
5 40059c: 91192000 add x0, x0, #0x648
6 4005a0: 97ffffa0 bl 400420 <puts@plt>
7 4005a4: 52800000 mov w0, #0x0 // #0
8 4005a8: a8c17bfd ldp x29, x30, [sp],#16
9 4005ac: d65f03c0 ret
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

10
11 ...
12
13 Contents of section .rodata:
14 400640 01000200 00000000 48656c6c 6f210a00 .....Hello!..

```

В ARM64 нет режима Thumb и Thumb-2, только ARM, так что тут только 32-битные инструкции.

Регистров тут в 2 раза больше: ?? (стр. ??). 64-битные регистры теперь имеют префикс X-, а их 32-битные части — W-.

Инструкция STP (*Store Pair*) сохраняет в стеке сразу два регистра: X29 и X30. Конечно, эта инструкция может сохранять эту пару где угодно в памяти, но здесь указан регистр **SP!**, так что пара сохраняется именно в стеке.

Регистры в ARM64 64-битные, каждый имеет длину в 8 байт, так что для хранения двух регистров нужно именно 16 байт.

Восклицательный знак (“!”) после операнда означает, что сначала от **SP!** будет отнято 16 и только затем значения из пары регистров будут записаны в стек.

Это называется *pre-index*. Больше о разнице между *post-index* и *pre-index* описано здесь: ?? (стр. ??).

Таким образом, в терминах более знакомого всем процессора x86, первая инструкция — это просто аналог пары инструкций PUSH X29 и PUSH X30. X29 в ARM64 используется как **FP!**<sup>41</sup>, а X30 как **LR!**, поэтому они сохраняются в прологе функции и восстанавливаются в эпилоге.

Вторая инструкция копирует **SP!** в X29 (или **FP!**). Это нужно для установки стекового фрейма функции.

Инструкции ADRP и ADD нужны для формирования адреса строки «Hello!» в регистре X0, ведь первый аргумент функции передается через этот регистр. Но в ARM нет инструкций, при помощи которых можно записать в регистр длинное число (потому что сама длина инструкции ограничена 4-я байтами. Больше об этом здесь: ?? (стр. ??)). Так что нужно использовать несколько инструкций. Первая инструкция (ADRP) записывает в X0 адрес 4-килобайтной страницы где находится строка, а вторая (ADD) просто прибавляет к этому адресу остаток. Читайте больше об этом: ?? (стр. ??).

$0x400000 + 0x648 = 0x400648$ , и мы видим, что в секции данных .rodata по этому адресу как раз находится наша Си-строка «Hello!».

Затем при помощи инструкции BL вызывается puts(). Это уже рассматривалось ранее: ?? (стр. ??).

Инструкция MOV записывает 0 в W0. W0 это младшие 32 бита 64-битного регистра X0:

<sup>41</sup>**FP!**

Старшие 32 бита	младшие 32 бита
X0	
	W0

А результат функции возвращается через X0, и `main()` возвращает 0, так что вот так готовится возвращаемый результат.

Почему именно 32-битная часть? Потому что в ARM64, как и в x86-64, тип `int` оставили 32-битным, для лучшей совместимости.

Следовательно, раз уж функция возвращает 32-битный `int`, то нужно заполнить только 32 младших бита регистра X0.

Для того, чтобы удостовериться в этом, немного отредактируем этот пример и перекомпилируем его.

Теперь `main()` возвращает 64-битное значение:

Листинг 1.30: `main()` возвращающая значение типа `uint64_t`

```
#include <stdio.h>
#include <stdint.h>

uint64_t main()
{
    printf ("Hello!\n");
    return 0;
}
```

Результат точно такой же, только MOV в той строке теперь выглядит так:

Листинг 1.31: Неоптимизирующий GCC 4.8.1 + objdump

```
4005a4:    d2800000    mov     x0, #0x0    // #0
```

Далее при помощи инструкции LDP (*Load Pair*) восстанавливаются регистры X29 и X30.

Восклицательного знака после инструкции нет. Это означает, что сначала значения достаются из стека, и только потом **SP!** увеличивается на 16.

Это называется *post-index*.

В ARM64 есть новая инструкция: RET. Она работает так же как и BX LR, но там добавлен специальный бит, подсказывающий процессору, что это именно выход из функции, а не просто переход, чтобы процессор мог более оптимально исполнять эту инструкцию.

Из-за простоты этой функции оптимизирующий GCC генерирует точно такой же код.

## 1.5.4. MIPS

### 0 «глобальном указателе» («global pointer»)

«Глобальный указатель» («global pointer») — это важная концепция в MIPS. Как

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

мы уже возможно знаем, каждая инструкция в MIPS имеет размер 32 бита, поэтому невозможно закодировать 32-битный адрес внутри одной инструкции. Вместо этого нужно использовать пару инструкций (как это сделал GCC для загрузки адреса текстовой строки в нашем примере). С другой стороны, используя только одну инструкцию, возможно загружать данные по адресам в пределах  $register - 32768 \dots register + 32767$ , потому что 16 бит знакового смещения можно закодировать в одной инструкции). Так мы можем выделить какой-то регистр для этих целей и ещё выделить буфер в 64KiB для самых часто используемых данных. Выделенный регистр называется «глобальный указатель» («global pointer») и он указывает на середину области 64KiB. Эта область обычно содержит глобальные переменные и адреса импортированных функций вроде `printf()`, потому что разработчики GCC решили, что получение адреса функции должно быть как можно более быстрой операцией, исполняющейся за одну инструкцию вместо двух. В ELF-файле эта 64KiB-область находится частично в секции `.sbss` («small **BSS!**<sup>42</sup>») для неинициализированных данных и в секции `.sdata` («small data») для инициализированных данных. Это значит что программист может выбирать, к чему нужен как можно более быстрый доступ, и затем расположить это в секциях `.sdata/.sbss`. Некоторые программисты «старой школы» могут вспомнить модель памяти в MS-DOS ?? (стр. ??) или в менеджерах памяти вроде XMS/EMS, где вся память делилась на блоки по 64KiB.

Эта концепция применяется не только в MIPS. По крайней мере PowerPC также использует эту технику.

### Оптимизирующий GCC

Рассмотрим следующий пример, иллюстрирующий концепцию «глобального указателя».

Листинг 1.32: Оптимизирующий GCC 4.4.5 (вывод на ассемблере)

```

1  $LC0:
2  ; \000 это ноль в восьмеричной системе:
3  .ascii "Hello, world!\012\000"
4  main:
5  ; пролог функции
6  ; установить GP:
7      lui    $28,%hi(__gnu_local_gp)
8      addiu  $sp,$sp,-32
9      addiu  $28,$28,%lo(__gnu_local_gp)
10 ; сохранить RA в локальном стеке:
11     sw     $31,28($sp)
12 ; загрузить адрес функции puts() из GP в $25:
13     lw     $25,%call16(puts)($28)
14 ; загрузить адрес текстовой строки в $4 ($a0):
15     lui    $4,%hi($LC0)
16 ; перейти на puts(), сохранив адрес возврата в link-регистре:
17     jalr   $25
18     addiu  $4,$4,%lo($LC0) ; branch delay slot

```

<sup>42</sup>**BSS!**

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

19 ; восстановить RA:
20     lw     $31,28($sp)
21 ; скопировать 0 из $zero в $v0:
22     move   $2,$0
23 ; вернуть управление сделав переход по адресу в RA:
24     j      $31
25 ; эпилог функции:
26     addiu  $sp,$sp,32 ; branch delay slot + освободить стек от локальных
        переменных

```

Как видно, регистр \$GP в прологе функции выставляется в середину этой области. Регистр **RA!** сохраняется в локальном стеке. Здесь также используется puts() вместо printf(). Адрес функции puts() загружается в \$25 инструкцией LW («Load Word»). Затем адрес текстовой строки загружается в \$4 парой инструкций LUI («Load Upper Immediate») и ADDIU («Add Immediate Unsigned Word»). LUI устанавливает старшие 16 бит регистра (поэтому в имени инструкции присутствует «upper») и ADDIU прибавляет младшие 16 бит к адресу. ADDIU следует за JALR (помните о *branch delay slots?*). Регистр \$4 также называется \$A0, который используется для передачи первого аргумента функции<sup>43</sup>. JALR («Jump and Link Register») делает переход по адресу в регистре \$25 (там адрес puts()) при этом сохраняя адрес следующей инструкции (LW) в **RA!**. Это так же как и в ARM. И ещё одна важная вещь: адрес сохраняемый в **RA!** это адрес не следующей инструкции (потому что это *delay slot* и исполняется перед инструкцией перехода), а инструкции после неё (после *delay slot*). Таким образом во время исполнения JALR в **RA!** записывается PC + 8. В нашем случае это адрес инструкции LW следующей после ADDIU.

LW («Load Word») в строке 20 восстанавливает **RA!** из локального стека (эта инструкция скорее часть эпилога функции).

MOVE в строке 22 копирует значение из регистра \$0 (\$ZERO) в \$2 (\$V0).

В MIPS есть *константный* регистр, всегда содержащий ноль. Должно быть, разработчики MIPS решили, что 0 это самая востребованная константа в программировании, так что пусть будет использоваться регистр \$0, всякий раз, когда будет нужен 0. Другой интересный факт: в MIPS нет инструкции, копирующей значения из регистра в регистр. На самом деле, MOVE DST, SRC это ADD DST, SRC, \$ZERO ( $DST = SRC + 0$ ), которая делает тоже самое. Очевидно, разработчики MIPS хотели сделать как можно более компактную таблицу опкодов. Это не значит, что сложение происходит во время каждой инструкции MOVE. Скорее всего, эти псевдоинструкции оптимизируются в **CPU!** и **ALU!**<sup>44</sup> никогда не используется.

J в строке 24 делает переход по адресу в **RA!**, и это работает как выход из функции. ADDIU после J на самом деле исполняется перед J (помните о *branch delay slots?*) и это часть эпилога функции.

Вот листинг сгенерированный **IDA!**. Каждый регистр имеет свой псевдоним:

### Листинг 1.33: Оптимизирующий GCC 4.4.5 (**IDA!**)

<sup>43</sup>Таблица регистров в MIPS доступна в приложении ?? (стр. ??)

<sup>44</sup>**ALU!**



```

1 .text:00000000 main:
2 .text:00000000
3 .text:00000000 var_10      = -0x10
4 .text:00000000 var_4      = -4
5 .text:00000000
6 ; пролог функции
7 ; установить GP:
8 .text:00000000          lui    $gp, (__gnu_local_gp >> 16)
9 .text:00000004          addiu   $sp, -0x20
10 .text:00000008         la     $gp, (__gnu_local_gp & 0xFFFF)
11 ; сохранить RA в локальном стеке:
12 .text:0000000C         sw     $ra, 0x20+var_4($sp)
13 ; сохранить GP в локальном стеке:
14 ; по какой-то причине, этой инструкции не было в ассемблерном выводе в GCC:
15 .text:00000010         sw     $gp, 0x20+var_10($sp)
16 ; загрузить адрес функции puts() из GP в $t9:
17 .text:00000014         lw     $t9, (puts & 0xFFFF)($gp)
18 ; сформировать адрес текстовой строки в $a0:
19 .text:00000018         lui    $a0, ($LC0 >> 16) # "Hello, world!"
20 ; перейти на puts(), сохранив адрес возврата в link-регистре:
21 .text:0000001C         jalr   $t9
22 .text:00000020         la     $a0, ($LC0 & 0xFFFF) # "Hello,
23     world!"
24 ; восстановить RA:
25 .text:00000024         lw     $ra, 0x20+var_4($sp)
26 ; скопировать 0 из $zero в $v0:
27 .text:00000028         move   $v0, $zero
28 ; вернуть управление сделав переход по адресу в RA:
29 .text:0000002C         jr     $ra
30 ; эпилог функции:
31 .text:00000030         addiu   $sp, 0x20

```

Инструкция в строке 15 сохраняет GP в локальном стеке. Эта инструкция мистическим образом отсутствует в листинге от GCC, может быть из-за ошибки в самом GCC<sup>45</sup>. Значение GP должно быть сохранено, потому что всякая функция может работать со своим собственным окном данных размером 64KiB. Регистр, содержащий адрес функции puts() называется \$T9, потому что регистры с префиксом T- называются «temporaries» и их содержимое можно не сохранять.

## Неоптимизирующий GCC

Неоптимизирующий GCC более многословный.

Листинг 1.34: Неоптимизирующий GCC 4.4.5 (вывод на ассемблере)

```

1 $LC0:
2     .ascii "Hello, world!\012\000"
3 main:
4 ; пролог функции
5 ; сохранить RA ($31) и FP в стеке:
6     addiu $sp,$sp,-32

```

<sup>45</sup>Очевидно, функция вывода листингов не так критична для пользователей GCC, поэтому там вполне могут быть неисправленные косметические ошибки.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

7      sw      $31,28($sp)
8      sw      $fp,24($sp)
9      ; установить FP (указатель стекового фрейма):
10     move   $fp,$sp
11     ; установить GP:
12     lui    $28,%hi(__gnu_local_gp)
13     addiu  $28,$28,%lo(__gnu_local_gp)
14     ; загрузить адрес текстовой строки:
15     lui    $2,%hi($LC0)
16     addiu  $4,$2,%lo($LC0)
17     ; загрузить адрес функции puts() используя GP:
18     lw     $2,%call16(puts)($28)
19     nop
20     ; вызвать puts():
21     move   $25,$2
22     jalr  $25
23     nop   ; branch delay slot
24
25     ; восстановить GP из локального стека:
26     lw     $28,16($fp)
27     ; установить регистр $2 ($V0) в ноль:
28     move   $2,$0
29     ; эпилог функции.
30     ; восстановить SP:
31     move   $sp,$fp
32     ; восстановить RA:
33     lw     $31,28($sp)
34     ; восстановить FP:
35     lw     $fp,24($sp)
36     addiu  $sp,$sp,32
37     ; переход на RA:
38     j     $31
39     nop   ; branch delay slot

```

Мы видим, что регистр FP используется как указатель на фрейм стека. Мы также видим 3 **NOPI**-а. Второй и третий следуют за инструкциями перехода. Видимо, компилятор GCC всегда добавляет **NOPI**-ы (из-за *branch delay slots*) после инструкций переходов и затем, если включена оптимизация, от них может избавляться. Так что они остались здесь.

Вот также листинг от **IDA!**:

Листинг 1.35: Неоптимизирующий GCC 4.4.5 (**IDA!**)

```

1  .text:00000000 main:
2  .text:00000000
3  .text:00000000 var_10      = -0x10
4  .text:00000000 var_8      = -8
5  .text:00000000 var_4      = -4
6  .text:00000000
7  ; пролог функции
8  ; сохранить RA и FP в стеке:
9  .text:00000000          addiu   $sp, -0x20
10 .text:00000004          sw      $ra, 0x20+var_4($sp)

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

11 .text:00000008      sw      $fp, 0x20+var_8($sp)
12 ; установить FP (указатель стекового фрейма):
13 .text:0000000C      move    $fp, $sp
14 ; установить GP:
15 .text:00000010      la      $gp, __gnu_local_gp
16 .text:00000018      sw      $gp, 0x20+var_10($sp)
17 ; загрузить адрес текстовой строки:
18 .text:0000001C      lui     $v0, (aHelloWorld >> 16) # "Hello,
19 | .text:00000020      addiu   $a0, $v0, (aHelloWorld & 0xFFFF) #
   | "Hello, world!"
20 ; загрузить адрес функции puts() используя GP:
21 .text:00000024      lw      $v0, (puts & 0xFFFF)($gp)
22 .text:00000028      or      $at, $zero ; NOP
23 ; вызвать puts():
24 .text:0000002C      move    $t9, $v0
25 .text:00000030      jalr   $t9
26 .text:00000034      or      $at, $zero ; NOP
27 ; восстановить GP из локального стека:
28 .text:00000038      lw      $gp, 0x20+var_10($fp)
29 ; установить регистр $2 ($V0) в ноль:
30 .text:0000003C      move    $v0, $zero
31 ; эпилог функции.
32 ; восстановить SP:
33 .text:00000040      move    $sp, $fp
34 ; восстановить RA:
35 .text:00000044      lw      $ra, 0x20+var_4($sp)
36 ; восстановить FP:
37 .text:00000048      lw      $fp, 0x20+var_8($sp)
38 .text:0000004C      addiu   $sp, 0x20
39 ; переход на RA:
40 .text:00000050      jr      $ra
41 .text:00000054      or      $at, $zero ; NOP

```

Интересно что **IDA!** распознала пару инструкций LUI/ADDIU и собрала их в одну псевдоинструкцию LA («Load Address») в строке 15. Мы также видим, что размер этой псевдоинструкции 8 байт! Это псевдоинструкция (или *макрос*), потому что это не настоящая инструкция MIPS, а скорее просто удобное имя для пары инструкций.

Ещё кое что: **IDA!** не распознала **NOP!**-инструкции в строках 22, 26 и 41.

Это OR \$AT, \$ZERO. По своей сути это инструкция, применяющая операцию *ИЛИ* к содержимому регистра \$AT с нулем, что, конечно же, холостая операция. MIPS, как и многие другие **ISA!**, не имеет отдельной **NOP!**-инструкции.

### Роль стекового фрейма в этом примере

Адрес текстовой строки передается в регистре. Так зачем устанавливать локальный стек? Причина в том, что значения регистров **RA!** и GP должны быть сохранены где-то (потому что вызывается printf()) и для этого используется локальный стек.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Если бы это была [leaf function](#), тогда можно было бы избавиться от пролога и эпилога функции. Например: ?? (стр. ??).

## Оптимизирующий GCC: загрузим в GDB

Листинг 1.36: пример сессии в GDB

```

root@debian-mips:~# gcc hw.c -O3 -o hw

root@debian-mips:~# gdb hw
GNU gdb (GDB) 7.0.1-debian
...
Reading symbols from /root/hw...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x400654
(gdb) run
Starting program: /root/hw

Breakpoint 1, 0x00400654 in main ()
(gdb) set step-mode on
(gdb) disas
Dump of assembler code for function main:
0x00400640 <main+0>:   lui    gp,0x42
0x00400644 <main+4>:   addiu  sp,sp,-32
0x00400648 <main+8>:   addiu  gp,gp,-30624
0x0040064c <main+12>:  sw     ra,28(sp)
0x00400650 <main+16>:  sw     gp,16(sp)
0x00400654 <main+20>:  lw     t9,-32716(gp)
0x00400658 <main+24>:  lui    a0,0x40
0x0040065c <main+28>:  jalr   t9
0x00400660 <main+32>:  addiu  a0,a0,2080
0x00400664 <main+36>:  lw     ra,28(sp)
0x00400668 <main+40>:  move   v0,zero
0x0040066c <main+44>:  jr     ra
0x00400670 <main+48>:  addiu  sp,sp,32
End of assembler dump.
(gdb) s
0x00400658 in main ()
(gdb) s
0x0040065c in main ()
(gdb) s
0x2ab2de60 in printf () from /lib/libc.so.6
(gdb) x/s $a0
0x400820:      "hello, world"
(gdb)

```

### 1.5.5. Вывод

Основная разница между кодом x86/ARM и x64/ARM64 в том, что указатель на строку теперь 64-битный. Действительно, ведь для того современные **CPU!** и стали 64-битными, потому что подешевела память, её теперь можно поставить

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

в компьютер намного больше, и чтобы её адресовать, 32-х бит уже недостаточно. Поэтому все указатели теперь 64-битные.

### 1.5.6. Упражнения

- <http://challenges.re/48>
- <http://challenges.re/49>

## 1.6. Пролог и эпилог функций

Пролог функции это инструкции в самом начале функции. Как правило, это что-то вроде такого фрагмента кода:

```
push    ebp
mov     ebp, esp
sub     esp, X
```

Эти инструкции делают следующее: сохраняют значение регистра EBP на будущее, выставляют EBP равным ESP, затем подготавливают место в стеке для хранения локальных переменных.

EBP сохраняет свое значение на протяжении всей функции, он будет использоваться здесь для доступа к локальным переменным и аргументам. Можно было бы использовать и ESP, но он постоянно меняется и это не очень удобно.

Эпилог функции аннулирует выделенное место в стеке, восстанавливает значение EBP на старое и возвращает управление в вызывающую функцию:

```
mov     esp, ebp
pop     ebp
ret     0
```

Пролог и эпилог функции обычно находятся в дизассемблерах для отделения функций друг от друга.

### 1.6.1. Рекурсия

Наличие эпилога и пролога может несколько ухудшить эффективность рекурсии.

Больше о рекурсии в этой книге: ?? (стр. ??).

## 1.7. Еще кое-что о пустой ф-ции

Вернемся к примеру с пустой ф-цией ?? (стр. ??). Теперь, когда мы уже знаем о прологе и эпилоге ф-ции, вот эта пустая ф-ция ?? (стр. ??) скомпилированная неоптимизирующим GCC:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Листинг 1.37: Неоптимизирующий GCC 8.2 x64 (вывод на ассемблере)

```
f:
    push    rbp
    mov     rbp, rsp
    nop
    pop     rbp
    ret
```

Это RET, но пролог и эпилог ф-ции, вероятно, не был сооптимизирован и остался как есть. NOP это, похоже, артефакт компилятора. Так или иначе, единственная здесь рабочая инструкция это RET. Остальные инструкции могут быть убраны (или сооптимизированы).

## 1.8. Еще кое-что о возвращаемых значениях

Теперь, когда мы уже знаем о прологе и эпилоге ф-ции, попробуем скомпилировать пример, возвращающий значение (?? (стр. ??), ?? (стр. ??)), используя неоптимизирующий GCC:

Листинг 1.38: Неоптимизирующий GCC 8.2 x64 (вывод на ассемблере)

```
f:
    push    rbp
    mov     rbp, rsp
    mov     eax, 123
    pop     rbp
    ret
```

Рабочие здесь инструкции это MOV и RET, остальные – пролог и эпилог.

## 1.9. Стек

Стек в компьютерных науках — это одна из наиболее фундаментальных структур данных <sup>46</sup>. **АКА!** <sup>47</sup> **LIFO!** <sup>48</sup>.

Технически это просто блок памяти в памяти процесса + регистр ESP в x86 или RSP в x64, либо **SP!** в ARM, который указывает где-то в пределах этого блока.

Часто используемые инструкции для работы со стеком — это PUSH и POP (в x86 и Thumb-режиме ARM). PUSH уменьшает ESP/RSP/**SP!** на 4 в 32-битном режиме (или на 8 в 64-битном), затем записывает по адресу, на который указывает ESP/RSP/**SP!**, содержимое своего единственного операнда.

POP это обратная операция — сначала достает из [указателя стека](#) значение и помещает его в операнд (который очень часто является регистром) и затем увеличивает указатель стека на 4 (или 8).

<sup>46</sup> [wikipedia.org/wiki/Call\\_stack](https://wikipedia.org/wiki/Call_stack)

<sup>47</sup> **АКА!**

<sup>48</sup> **LIFO!**

В самом начале **регистр-указатель** указывает на конец стека. Конец стека находится в начале блока памяти, выделенного под стек. Это странно, но это так. PUSH уменьшает **регистр-указатель**, а POP — увеличивает.

В процессоре ARM, тем не менее, есть поддержка стеков, растущих как в сторону уменьшения, так и в сторону увеличения.

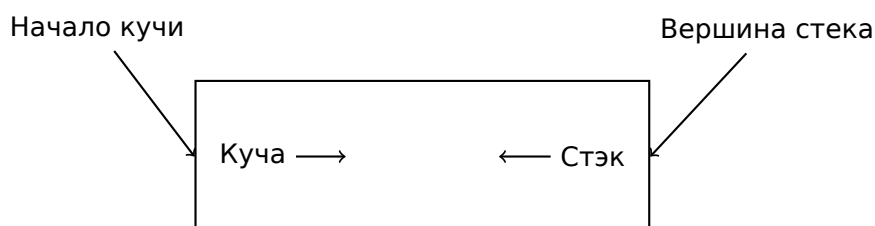
Например, инструкции **STMGD!/LDMGD!**, **STMED!**<sup>49</sup>/**LDMED!**<sup>50</sup> предназначены для descending-стека (растет назад, начиная с высоких адресов в сторону низких).

Инструкции **STMFA!**<sup>51</sup>/**LDMFA!**<sup>52</sup>, **STMEA!**<sup>53</sup>/**LDMEA!**<sup>54</sup> предназначены для ascending-стека (растет вперед, начиная с низких адресов в сторону высоких).

### 1.9.1. Почему стек растет в обратную сторону?

Интуитивно мы можем подумать, что, как и любая другая структура данных, стек мог бы расти вперед, т.е. в сторону увеличения адресов.

Причина, почему стек растет назад, видимо, историческая. Когда компьютеры были большие и занимали целую комнату, было очень легко разделить сегмент на две части: для **кучи** и для стека. Заранее было неизвестно, насколько большой может быть **куча** или стек, так что это решение было самым простым.



В [D. M. Ritchie and K. Thompson, *The UNIX Time Sharing System*, (1974)]<sup>55</sup> можно прочитать:

The user-core part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual address space. During execution, this segment is write-protected and a single copy of it is shared among all processes executing the same program. At the first 8K byte boundary above the program text segment in the virtual address space begins a nonshared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a stack

<sup>49</sup>**STMED!**

<sup>50</sup>**LDMED!**

<sup>51</sup>**STMFA!**

<sup>52</sup>**LDMFA!**

<sup>53</sup>**STMEA!**

<sup>54</sup>**LDMEA!**

<sup>55</sup>Также доступно здесь: [URL](#)

segment, which automatically grows downward as the hardware's stack pointer fluctuates.

Это немного напоминает как некоторые студенты пишут два конспекта в одной тетрадке: первый конспект начинается обычным образом, второй пишется с конца, перевернув тетрадку. Конспекты могут встретиться где-то посередине, в случае недостатка свободного места.

## 1.9.2. Для чего используется стек?

### Сохранение адреса возврата управления

#### x86

При вызове другой функции через CALL сначала в стек записывается адрес, указывающий на место после инструкции CALL, затем делается безусловный переход (почти как JMP) на адрес, указанный в операнде.

CALL — это аналог пары инструкций PUSH `address_after_call` / JMP.

RET вытаскивает из стека значение и передает управление по этому адресу — это аналог пары инструкций POP `tmp` / JMP `tmp`.

Крайне легко устроить переполнение стека, запустив бесконечную рекурсию:

```
void f()
{
    f();
};
```

MSVC 2008 предупреждает о проблеме:

```
c:\tmp6>cl ss.cpp /Fass.asm
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for x86
Copyright (C) Microsoft Corporation. All rights reserved.

ss.cpp
c:\tmp6\ss.cpp(4) : warning C4717: 'f' : recursive on all control paths,
function will cause runtime stack overflow
```

...но, тем не менее, создает нужный код:

```
?f@YAXXZ PROC                ; f
; Line 2
    push    ebp
    mov     ebp, esp
; Line 3
    call   ?f@YAXXZ           ; f
; Line 4
    pop     ebp
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

ret    0
?f@YAXXZ ENDP          ; f

```

...причем, если включить оптимизацию (/Ox), то будет даже интереснее, без переполнения стека, но работать будет *корректно*<sup>56</sup>:

```

?f@YAXXZ PROC          ; f
; Line 2
$LL3@f:
; Line 3
        jmp     SHORT $LL3@f
?f@YAXXZ ENDP          ; f

```

GCC 4.4.1 генерирует точно такой же код в обоих случаях, хотя и не предупреждает о проблеме.

## ARM

Программы для ARM также используют стек для сохранения **RA!**, куда нужно вернуться, но несколько иначе. Как уже упоминалось в секции «Hello, world!» (?? (стр. ??)), **RA!** записывается в регистр **LR!** ([link register](#)). Но если есть необходимость вызывать какую-то другую функцию и использовать регистр **LR!** ещё раз, его значение желательно сохранить.

Обычно это происходит в прологе функции, часто мы видим там инструкцию вроде PUSH {R4-R7, LR}, а в эпилоге POP {R4-R7, PC} — так сохраняются регистры, которые будут использоваться в текущей функции, в том числе **LR!**.

Тем не менее, если некая функция не вызывает никаких более функций, в терминологии **RISC!** она называется *leaf function*<sup>57</sup>. Как следствие, «leaf»-функция не сохраняет регистр **LR!** (потому что не изменяет его). А если эта функция небольшая, использует мало регистров, она может не использовать стек вообще. Таким образом, в ARM возможен вызов небольших leaf-функций не используя стек. Это может быть быстрее чем в старых x86, ведь внешняя память для стека не используется<sup>58</sup>. Либо это может быть полезным для тех ситуаций, когда память для стека ещё не выделена, либо недоступна,

Некоторые примеры таких функций: ?? (стр. ??), ?? (стр. ??), ?? (стр. ??), ?? (стр. ??), ?? (стр. ??), ?? (стр. ??), ?? (стр. ??), ?? (стр. ??).

## Передача параметров функции

Самый распространенный способ передачи параметров в x86 называется «cdecl»:

<sup>56</sup>Здесь ирония

<sup>57</sup>[infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html](http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html)

<sup>58</sup>Когда-то, очень давно, на PDP-11 и VAX на инструкцию CALL (вызов других функций) могло тратиться вплоть до 50% времени (возможно из-за работы с памятью), поэтому считалось, что много небольших функций это [анти-паттерн](#) [Eric S. Raymond, *The Art of UNIX Programming*, (2003)Chapter 4, Part II].

```

push arg3
push arg2
push arg1
call f
add esp, 12 ; 4*3=12

```

Вызываемая функция получает свои параметры также через указатель стека. Следовательно, так расположены значения в стеке перед исполнением самой первой инструкции функции `f()`:

ESP	адрес возврата
ESP+4	аргумент#1, маркируется в <b>IDA!</b> как <code>arg_0</code>
ESP+8	аргумент#2, маркируется в <b>IDA!</b> как <code>arg_4</code>
ESP+0xC	аргумент#3, маркируется в <b>IDA!</b> как <code>arg_8</code>
...	...

См. также в соответствующем разделе о других способах передачи аргументов через стек (?? (стр. ??)).

Кстати, вызываемая функция не имеет информации о количестве переданных ей аргументов. Функции Си с переменным количеством аргументов (как `printf()`) могут определять их количество по спецификаторам строки формата (начинающиеся со знака %).

Если написать что-то вроде:

```
printf("%d %d %d", 1234);
```

`printf()` выведет 1234, затем ещё два случайных числа<sup>59</sup>, которые волею случая оказались в стеке рядом.

Вот почему не так уж и важно, как объявлять функцию `main()`: как `main()`, `main(int argc, char *argv[])` либо `main(int argc, char *argv[], char *envp[])`.

В реальности, **CRT!**-код вызывает `main()` примерно так:

```

push envp
push argv
push argc
call main
...

```

Если вы объявляете `main()` без аргументов, они, тем не менее, присутствуют в стеке, но не используются. Если вы объявите `main()` как `main(int argc, char *argv[])`, вы можете использовать два первых аргумента, а третий останется для вашей функции «невидимым». Более того, можно даже объявить `main(int argc)`, и это будет работать.

<sup>59</sup>В строгом смысле, они не случайны, скорее, непредсказуемы: ?? (стр. ??)

## Альтернативные способы передачи аргументов

Важно отметить, что, в общем, никто не заставляет программистов передавать параметры именно через стек, это не является требованием к исполняемому коду. Вы можете делать это совершенно иначе, не используя стек вообще.

В каком-то смысле, популярный метод среди начинающих использовать язык ассемблера, это передавать аргументы в глобальных переменных, например:

Листинг 1.39: Код на ассемблере

```

...
mov     X, 123
mov     Y, 456
call    do_something

...
X       dd     ?
Y       dd     ?

do_something proc near
; take X
; take Y
; do something
retn
do_something endp

```

Но у этого метода есть очевидный недостаток: ф-ция *do\_something()* не сможет вызвать саму себя рекурсивно (либо, через какую-то стороннюю ф-цию), потому что тогда придется затереть свои собственные аргументы. Та же история с локальными переменными: если хранить их в глобальных переменных, ф-ция не сможет вызывать сама себя. К тому же, этот метод не безопасный для мультитредовой среды<sup>60</sup>. Способ хранения подобной информации в стеке заметно всё упрощает — он может хранить столько аргументов ф-ций и/или значений вообще, сколько в нем есть места.

В [Donald E. Knuth, *The Art of Computer Programming*, Volume 1, 3rd ed., (1997), 189] можно прочитать про еще более странные схемы передачи аргументов, которые были очень удобны на IBM System/360.

В MS-DOS был метод передачи аргументов через регистры, например, этот фрагмент кода для древней 16-битной MS-DOS выводит "Hello, world!":

```

mov  dx, msg      ; адрес сообщения
mov  ah, 9        ; 9 означает ф-цию "вывод строки"
int  21h          ; DOS "syscall"

mov  ah, 4ch      ; ф-ция "закончить программу"

```

<sup>60</sup>При корректной реализации, каждый тред будет иметь свой собственный стек со своими аргументами/переменными.

```
int 21h          ; DOS "syscall"

msg db 'Hello, World!\$'
```

Это очень похоже на метод ?? (стр. ??). И еще на метод вызовов сисколлов в Linux (?? (стр. ??)) и Windows.

Если ф-ция в MS-DOS возвращает булево значение (т.е., один бит, обычно сигнализирующий об ошибке), часто использовался флаг CF.

Например:

```
mov ah, 3ch      ; создать файл
lea dx, filename
mov cl, 1
int 21h
jc error
mov file_handle, ax
...
error:
...
```

В случае ошибки, флаг CF будет выставлен. Иначе, хэндл только что созданного файла возвращается в AX.

Этот метод до сих пор используется программистами на ассемблере. В исходных кодах Windows Research Kernel (который очень похож на Windows 2003) мы можем найти такое (файл *base/ntos/ke/i386/cpu.asm*):

```
public Get386Stepping
Get386Stepping proc

    call    MultiplyTest          ; Perform multiplication test
    jnc     short G3s00           ; if nc, muttest is ok
    mov     ax, 0
    ret

G3s00:
    call    Check386B0           ; Check for B0 stepping
    jnc     short G3s05         ; if nc, it's B1/later
    mov     ax, 100h            ; It is B0/earlier stepping
    ret

G3s05:
    call    Check386D1           ; Check for D1 stepping
    jc      short G3s10         ; if c, it is NOT D1
    mov     ax, 301h            ; It is D1/later stepping
    ret

G3s10:
    mov     ax, 101h            ; assume it is B1 stepping
    ret

...
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

MultiplyTest    proc

    xor    cx,cx                ; 64K times is a nice round number
mlt00:    push   cx
           call  Multiply        ; does this chip's multiply work?
           pop   cx
           jc   short mltx       ; if c, No, exit
           loop mlt00            ; if nc, YEs, loop to try again
           clc
mltx:
           ret
MultiplyTest    endp

```

### Хранение локальных переменных

Функция может выделить для себя некоторое место в стеке для локальных переменных, просто отодвинув [указатель стека](#) глубже к концу стека.

Это очень быстро вне зависимости от количества локальных переменных. Хранить локальные переменные в стеке не является необходимым требованием. Вы можете хранить локальные переменные где угодно. Но по традиции всё сложилось так.

### x86: Функция `alloca()`

Интересен случай с функцией `alloca()` <sup>61</sup>. Эта функция работает как `malloc()`, но выделяет память прямо в стеке. Память освобождают через `free()` не нужно, так как эпилог функции (`??` (стр. `??`)) вернет ESP в изначальное состояние и выделенная память просто *выкидывается*. Интересна реализация функции `alloca()`. Эта функция, если упрощенно, просто сдвигает ESP вглубь стека на столько байт, делая так, что ESP указывает на *выделенный* блок.

Попробуем:

```

#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    sprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else

```

<sup>61</sup>В MSVC, реализацию функции можно посмотреть в файлах `alloca16.asm` и `chkstk.asm` в `C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\crt\src\intel`

```

    __snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};

```

Функция `__snprintf()` работает так же, как и `printf()`, только вместо выдачи результата в `stdout` (т.е. на терминал или в консоль), записывает его в буфер `buf`. Функция `puts()` выдает содержимое буфера `buf` в `stdout`. Конечно, можно было бы заменить оба этих вызова на один `printf()`, но здесь нужно проиллюстрировать использование небольшого буфера.

## MSVC

Компилируем (MSVC 2010):

Листинг 1.40: MSVC 2010

```

...

    mov     eax, 600 ; 00000258H
    call   __alloca_probe_16
    mov     esi, esp

    push   3
    push   2
    push   1
    push   OFFSET $SG2672
    push   600 ; 00000258H
    push   esi
    call   __snprintf

    push   esi
    call   _puts
    add    esp, 28

...

```

Единственный параметр в `alloca()` передается через EAX, а не как обычно через стек <sup>62</sup>.

## GCC + Синтаксис Intel

А GCC 4.4.1 обходится без вызова других функций:

<sup>62</sup>Это потому, что `alloca()` — это не сколько функция, сколько т.н. *compiler intrinsic* (?? (стр. ??))  
 Одна из причин, почему здесь нужна именно функция, а не несколько инструкций прямо в коде в том, что в реализации функции `alloca()` от **MSVC!**<sup>63</sup> есть также код, читающий из только что выделенной памяти, чтобы **OS!** подключила физическую память к этому региону **VM!**<sup>64</sup>. После вызова `alloca()` ESP указывает на блок в 600 байт, который мы можем использовать под `buf`.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Листинг 1.41: GCC 4.7.3

```

.LC0:
.string "hi! %d, %d, %d\n"
f:
    push    ebp
    mov     ebp, esp
    push    ebx
    sub     esp, 660
    lea    ebx, [esp+39]
    and     ebx, -16           ; выровнять указатель по 16-байтной
    границе
    mov     DWORD PTR [esp], ebx ; s
    mov     DWORD PTR [esp+20], 3
    mov     DWORD PTR [esp+16], 2
    mov     DWORD PTR [esp+12], 1
    mov     DWORD PTR [esp+8], OFFSET FLAT:.LC0 ; "hi! %d, %d, %d\n"
    mov     DWORD PTR [esp+4], 600 ; maxlen
    call    _sprintf
    mov     DWORD PTR [esp], ebx ; s
    call    puts
    mov     ebx, DWORD PTR [ebp-4]
    leave
    ret

```

**GCC + Синтаксис AT&T**

Посмотрим на тот же код, только в синтаксисе AT&T:

Листинг 1.42: GCC 4.7.3

```

.LC0:
.string "hi! %d, %d, %d\n"
f:
    pushl   %ebp
    movl   %esp, %ebp
    pushl   %ebx
    subl   $660, %esp
    leal   39(%esp), %ebx
    andl   $-16, %ebx
    movl   %ebx, (%esp)
    movl   $3, 20(%esp)
    movl   $2, 16(%esp)
    movl   $1, 12(%esp)
    movl   $.LC0, 8(%esp)
    movl   $600, 4(%esp)
    call   _sprintf
    movl   %ebx, (%esp)
    call   puts
    movl   -4(%ebp), %ebx
    leave
    ret

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Всё то же самое, что и в прошлом листинге.

Кстати, `movl $3, 20(%esp)` — это аналог `mov DWORD PTR [esp+20], 3` в синтаксисе Intel. Адресация памяти в виде *регистр+смещение* записывается в синтаксисе AT&T как смещение(`%регистр`).

### (Windows) SEH

В стеке хранятся записи **SEH!**<sup>65</sup> для функции (если они присутствуют). Читайте больше о нем здесь: (?? (стр. ??)).

### Защита от переполнений буфера

Здесь больше об этом (?? (стр. ??)).

### Автоматическое освобождение данных в стеке

Возможно, причина хранения локальных переменных и SEH-записей в стеке в том, что после выхода из функции, всё эти данные освобождаются автоматически, используя только одну инструкцию корректирования указателя стека (часто это `ADD`). Аргументы функций, можно сказать, тоже освобождаются автоматически в конце функции. А всё что хранится в куче (*heap*) нужно освободить явно.

### 1.9.3. Разметка типичного стека

Разметка типичного стека в 32-битной среде перед исполнением самой первой инструкции функции выглядит так:

...	...
ESP-0xC	локальная переменная#2, маркируется в <b>IDA!</b> как <code>var_8</code>
ESP-8	локальная переменная#1, маркируется в <b>IDA!</b> как <code>var_4</code>
ESP-4	сохраненное значениеEBP
ESP	Адрес возврата
ESP+4	аргумент#1, маркируется в <b>IDA!</b> как <code>arg_0</code>
ESP+8	аргумент#2, маркируется в <b>IDA!</b> как <code>arg_4</code>
ESP+0xC	аргумент#3, маркируется в <b>IDA!</b> как <code>arg_8</code>
...	...

### 1.9.4. Мусор в стеке

When one says that something seems random, what one usually means in practice is that one cannot see any regularities in it.

Stephen Wolfram, A New Kind of Science.

Часто в этой книге говорится о «шуме» или «мусоре» в стеке или памяти. Отку-

<sup>65</sup> **SEH!**

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



да он берется? Это то, что осталось там после исполнения предыдущих функций.

Короткий пример:

```
#include <stdio.h>

void f1()
{
    int a=1, b=2, c=3;
};

void f2()
{
    int a, b, c;
    printf ("%d, %d, %d\n", a, b, c);
};

int main()
{
    f1();
    f2();
};
```

Компилируем...

Листинг 1.43: Неоптимизирующий MSVC 2010

```
$SG2752 DB      '%d, %d, %d', 0aH, 00H

_c$ = -12      ; size = 4
_b$ = -8      ; size = 4
_a$ = -4      ; size = 4
_f1          PROC
    push      ebp
    mov       ebp, esp
    sub       esp, 12
    mov       DWORD PTR _a$[ebp], 1
    mov       DWORD PTR _b$[ebp], 2
    mov       DWORD PTR _c$[ebp], 3
    mov       esp, ebp
    pop       ebp
    ret       0
_f1          ENDP

_c$ = -12      ; size = 4
_b$ = -8      ; size = 4
_a$ = -4      ; size = 4
_f2          PROC
    push      ebp
    mov       ebp, esp
    sub       esp, 12
    mov       eax, DWORD PTR _c$[ebp]
    push     eax
    mov       ecx, DWORD PTR _b$[ebp]
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        push    ecx
        mov     edx, DWORD PTR _a$[ebp]
        push    edx
        push    OFFSET $SG2752 ; '%d, %d, %d'
        call   DWORD PTR __imp__printf
        add     esp, 16
        mov     esp, ebp
        pop     ebp
        ret     0
_f2     ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
        call   _f1
        call   _f2
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP

```

Компилятор поворчит немного...

```

c:\Polygon>cl st.c /Fast.asm /MD
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.40219.01 for x
  ↳ 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

st.c
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'c' x
  ↳ used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'b' x
  ↳ used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'a' x
  ↳ used
Microsoft (R) Incremental Linker Version 10.00.40219.01
Copyright (C) Microsoft Corporation. All rights reserved.

/out:st.exe
st.obj

```

Но когда мы запускаем...

```

c:\Polygon\c>st
1, 2, 3

```

Ох. Вот это странно. Мы ведь не устанавливали значения никаких переменных в f2(). Эти значения — это «привидения», которые всё ещё в стеке.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Загрузим пример в OllyDbg:

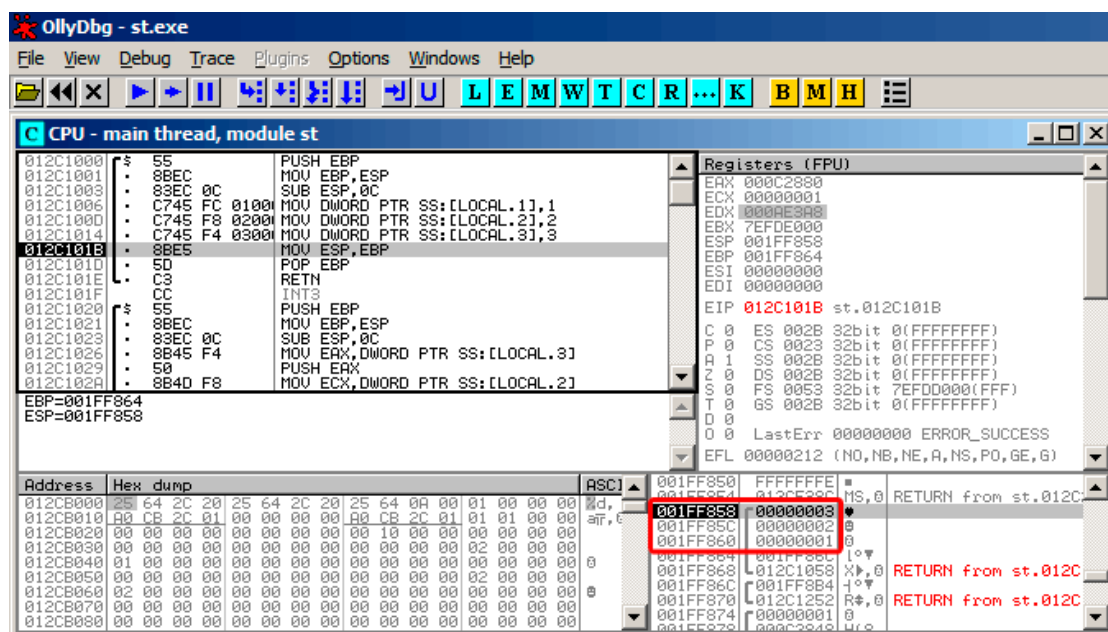


Рис. 1.6: OllyDbg: f1()

Когда `f1()` заполняет переменные `a`, `b` и `c` они сохраняются по адресу `0x1FF860`, и т. д.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

А когда выполняется `f2()`:

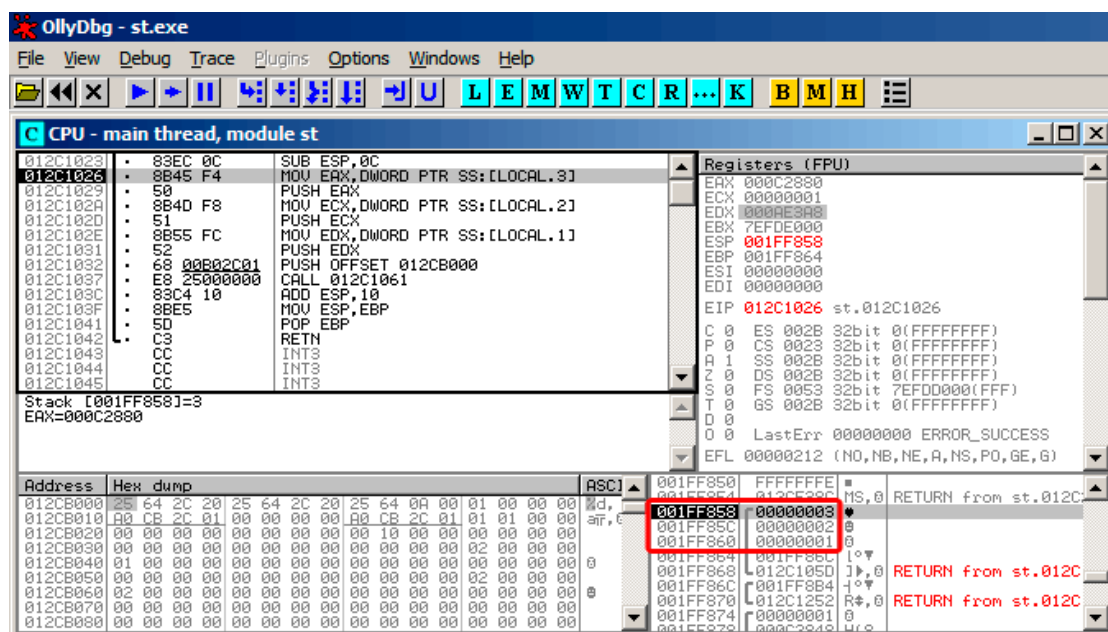


Рис. 1.7: OllyDbg: `f2()`

... *a*, *b* и *c* в функции `f2()` находятся по тем же адресам! Пока никто не перезаписал их, так что они здесь в нетронутом виде. Для создания такой странной ситуации несколько функций должны выполняться друг за другом и **SP!** должен быть одинаковым при входе в функции, т.е. у функций должно быть равное количество аргументов). Тогда локальные переменные будут расположены в том же месте стека. Подводя итоги, все значения в стеке (да и памяти вообще) это значения оставшиеся от исполнения предыдущих функций. Строго говоря, они не случайны, они скорее непредсказуемы. А как иначе? Можно было бы очищать части стека перед исполнением каждой функции, но это слишком много лишней (и ненужной) работы.

### MSVC 2013

Этот пример был скомпилирован в MSVC 2010. Но один читатель этой книги сделал попытку скомпилировать пример в MSVC 2013, запустил и увидел 3 числа в обратном порядке:

```
c:\Polygon\c>st
3, 2, 1
```

Почему? Я также попробовал скомпилировать этот пример в MSVC 2013 и увидел это:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Листинг 1.44: MSVC 2013

```

_a$ = -12      ; size = 4
_b$ = -8       ; size = 4
_c$ = -4       ; size = 4
_f2   PROC

...

_f2   ENDP

_c$ = -12      ; size = 4
_b$ = -8       ; size = 4
_a$ = -4       ; size = 4
_f1   PROC

...

_f1   ENDP

```

В отличие от MSVC 2010, MSVC 2013 разместил переменные a/b/c в функции f2() в обратном порядке. И это полностью корректно, потому что в стандартах Си/Си++ нет правила, в каком порядке локальные переменные должны быть размещены в локальном стеке, если вообще. Разница есть из-за того что MSVC 2010 делает это одним способом, а в MSVC 2013, вероятно, что-то немного изменили во внутренностях компилятора, так что он ведет себя слегка иначе.

### 1.9.5. Упражнения

- <http://challenges.re/51>
- <http://challenges.re/52>

## 1.10. Почти пустая ф-ция

Это фрагмент реального кода найденного в Boolector<sup>66</sup>:

```

// forward declaration. the function is residing in some other module:
int boolector_main (int argc, char **argv);

// executable
int main (int argc, char **argv)
{
    return boolector_main (argc, argv);
}

```

Зачем кому-то это делать? Не ясно, но можно предположить что boolector\_main() может быть скомпилирована в виде DLL или динамической библиотеки, и может вызываться во время тестов. Конечно, тесты могут подготовить переменные argc/argv так же, как это сделал бы **CRT!**

<sup>66</sup><https://boolector.github.io/>

Интересно, как это компилируется:

Листинг 1.45: Неоптимизирующий GCC 8.2 x64 (вывод на ассемблере)

```
main:
    push    rbp
    mov     rbp, rsp
    sub    rsp, 16
    mov     DWORD PTR -4[rbp], edi
    mov     QWORD PTR -16[rbp], rsi
    mov     rdx, QWORD PTR -16[rbp]
    mov     eax, DWORD PTR -4[rbp]
    mov     rsi, rdx
    mov     edi, eax
    call   boolector_main
    leave
    ret
```

Здесь у нас: пролог, ненужная (не соптимизированная) перетасовка двух аргументов, CALL, эпилог, RET. Посмотрим на оптимизированную версию:

Листинг 1.46: Оптимизирующий GCC 8.2 x64 (вывод на ассемблере)

```
main:
    jmp     boolector_main
```

Вот так просто: стек/регистры остаются как есть, а ф-ция `boolector_main()` имеет тот же набор аргументов. Так что всё что нужно, это просто передать управление по другому адресу.

Это близко к [thunk function](#).

Кое-что посложнее мы увидим позже: ?? (стр. ??), ?? (стр. ??).

## 1.11. printf() с несколькими аргументами

Попробуем теперь немного расширить пример *Hello, world!* (?? (стр. ??)), написав в теле функции `main()`:

```
#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d", 1, 2, 3);
    return 0;
};
```

### 1.11.1. x86

**x86: 3 целочисленных аргумента**

**MSVC**

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Компилируем при помощи MSVC 2010 Express, и в итоге получим:

```
$SG3830 DB      'a=%d; b=%d; c=%d', 00H
...
        push    3
        push    2
        push    1
        push    OFFSET $SG3830
        call    _printf
        add     esp, 16
```

Всё почти то же, за исключением того, что теперь видно, что аргументы для `printf()` заталкиваются в стек в обратном порядке: самый первый аргумент заталкивается последним.

Кстати, вспомним, что переменные типа *int* в 32-битной системе, как известно, имеют ширину 32 бита, это 4 байта.

Итак, у нас всего 4 аргумента.  $4 * 4 = 16$  — именно 16 байт занимают в стеке указатель на строку плюс ещё 3 числа типа *int*.

Когда при помощи инструкции `ADD ESP, X` корректируется [указатель стека](#) ESP после вызова какой-либо функции, зачастую можно сделать вывод о том, сколько аргументов у вызываемой функции было, разделив X на 4.

Конечно, это относится только к `cdecl`-методу передачи аргументов через стек, и только для 32-битной среды.

См. также в соответствующем разделе о способах передачи аргументов через стек ([?? \(стр. ??\)](#)).

Иногда бывает так, что подряд идут несколько вызовов разных функций, но стек корректируется только один раз, после последнего вызова:

```
push a1
push a2
call ...
...
push a1
call ...
...
push a1
push a2
push a3
call ...
add esp, 24
```

Вот пример из реальной жизни:

Листинг 1.47: x86

```
.text:100113E7  push    3
.text:100113E9  call    sub_100018B0 ; берет один аргумент (3)
.text:100113EE  call    sub_100019D0 ; не имеет аргументов вообще
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
.text:100113F3  call    sub_10006A90 ; не имеет аргументов вообще
.text:100113F8  push   1
.text:100113FA  call    sub_100018B0 ; берет один аргумент (1)
.text:100113FF  add     esp, 8      ; выбрасывает из стека два аргумента
```



## MSVC и OllyDbg

Попробуем этот же пример в OllyDbg. Это один из наиболее популярных win32-отладчиков пользовательского режима. Мы можем компилировать наш пример в MSVC 2012 с опцией /MD что означает линковать с библиотекой MSVCRT\*.DLL, чтобы импортируемые функции были хорошо видны в отладчике.

Затем загружаем исполняемый файл в OllyDbg. Самая первая точка останова в ntdll.dll, нажмите F9 (запустить). Вторая точка останова в **CRT!**-коде. Теперь мы должны найти функцию main().

Найдите этот код, прокрутив окно кода до самого верха (MSVC располагает функцию main() в самом начале секции кода):

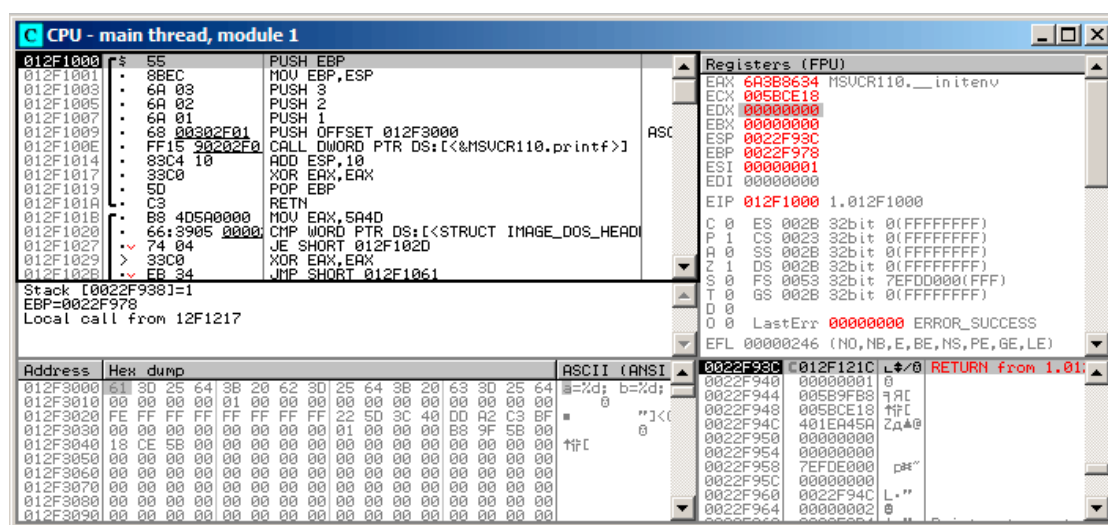


Рис. 1.8: OllyDbg: самое начало функции main()

Кликните на инструкции PUSH EBP, нажмите F2 (установка точки останова) и нажмите F9 (запустить). Нам нужно произвести все эти манипуляции, чтобы пропустить **CRT!**-код, потому что нам он пока не интересен.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Нажмите F8 (сделать шаг, не входя в функцию) 6 раз, т.е. пропустить 6 инструкций:

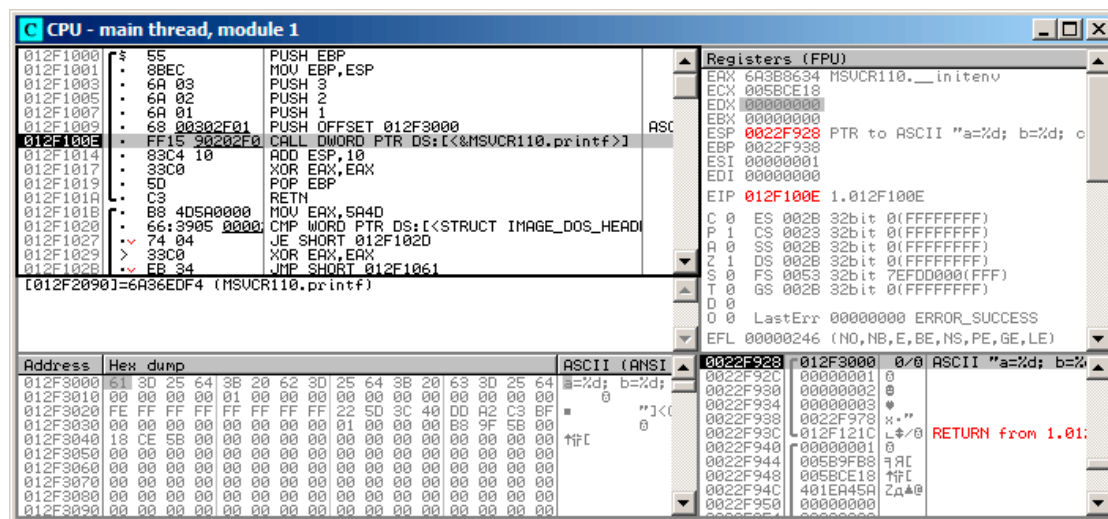


Рис. 1.9: OllyDbg: перед исполнением printf()

Теперь **PC!** указывает на инструкцию CALL printf. OllyDbg, как и другие отладчики, подсвечивает регистры со значениями, которые изменились. Поэтому каждый раз когда мы нажимаем F8, EIP изменяется и его значение подсвечивается красным. ESP также меняется, потому что значения заталкиваются в стек.

Где находятся эти значения в стеке? Посмотрите на правое нижнее окно в отладчике:

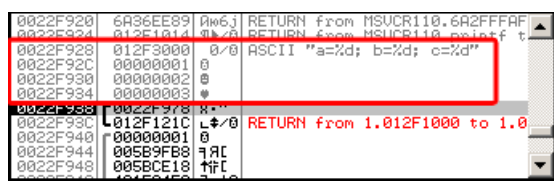


Рис. 1.10: OllyDbg: стек с сохраненными значениями (красная рамка добавлена в графическом редакторе)

Здесь видно 3 столбца: адрес в стеке, значение в стеке и ещё дополнительный комментарий от OllyDbg. OllyDbg может находить указатели на ASCII-строки в стеке, так что он показывает здесь printf()-строку.

Можно кликнуть правой кнопкой мыши на строке формата, кликнуть на «Follow in dump» и строка формата появится в окне слева внизу, где всегда виден

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

---

какой-либо участок памяти. Эти значения в памяти можно редактировать. Можно изменить саму строку формата, и тогда результат работы нашего примера будет другой. В данном случае пользы от этого немного, но для упражнения это полезно, чтобы начать чувствовать как тут всё работает.

Нажмите F8 (сделать шаг, не входя в функцию).

В консоли мы видим вывод:

```
a=1; b=2; c=3
```

Посмотрим как изменились регистры и состояние стека:

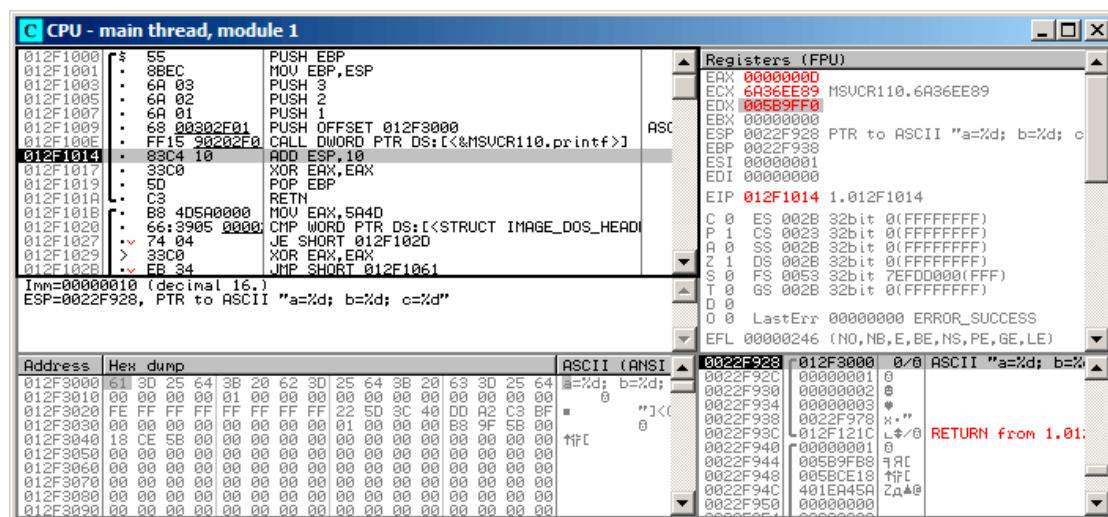


Рис. 1.11: OllyDbg после исполнения printf()

Регистр EAX теперь содержит 0xD (13). Всё верно: printf() возвращает количество выведенных символов. Значение EIP изменилось. Действительно, теперь здесь адрес инструкции после CALL printf. Значения регистров ECX и EDX также изменились. Очевидно, внутренности функции printf() используют их для каких-то своих нужд.

Очень важно то, что значение ESP не изменилось. И аргументы-значения в стеке также! Мы ясно видим здесь и строку формата и соответствующие ей 3 значения, они всё ещё здесь. Действительно, по соглашению вызовов cdecl, вызываемая функция не возвращает ESP назад. Это должна делать вызывающая функция (caller).

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Нажмите F8 снова, чтобы исполнилась инструкция ADD ESP, 10:

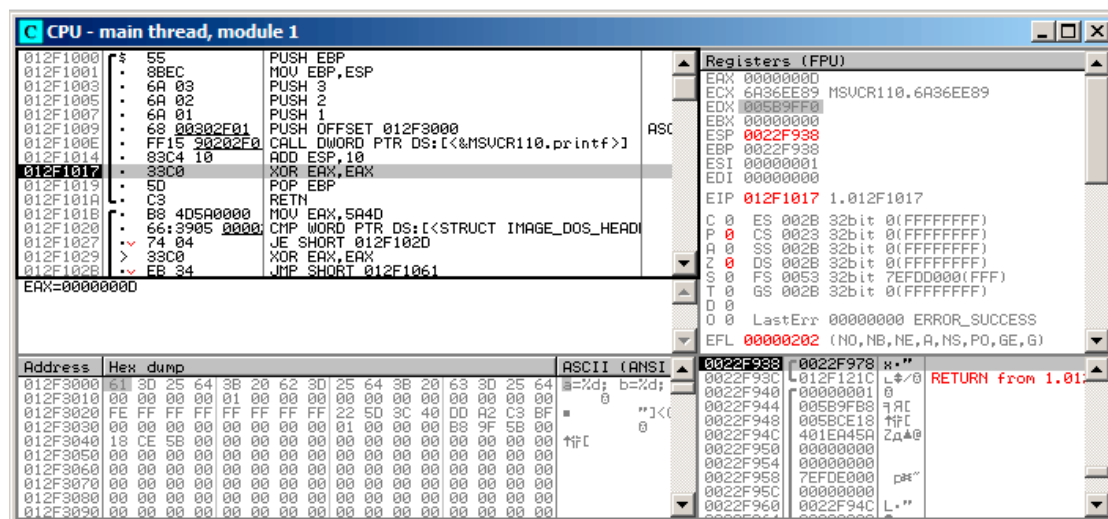


Рис. 1.12: OllyDbg: после исполнения инструкции ADD ESP, 10

ESP изменился, но значения всё ещё в стеке! Конечно, никому не нужно заполнять эти значения нулями или что-то в этом роде. Всё что выше указателя стека (**SP!**) это *шум* или *мусор* и не имеет особой ценности. Было бы очень затратно по времени очищать ненужные элементы стека, к тому же, никому это и не нужно.

## GCC

Скомпилируем то же самое в Linux при помощи GCC 4.4.1 и посмотрим на результат в **IDA!**:

```
main      proc near
var_10    = dword ptr -10h
var_C     = dword ptr -0Ch
var_8     = dword ptr -8
var_4     = dword ptr -4

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFFF0h
        sub     esp, 10h
        mov     eax, offset aADBDCD ; "a=%d; b=%d; c=%d"
        mov     [esp+10h+var_4], 3
        mov     [esp+10h+var_8], 2
        mov     [esp+10h+var_C], 1
        mov     [esp+10h+var_10], eax
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        call    _printf
        mov     eax, 0
        leave
        retn
main    endp

```

Можно сказать, что этот короткий код, созданный GCC, отличается от кода MSVC только способом помещения значений в стек. Здесь GCC снова работает со стеком напрямую без PUSH/POP.

## GCC и GDB

Попробуем также этот пример и в **GDB!**<sup>67</sup> в Linux.

-g означает генерировать отладочную информацию в выходном исполняемом файле.

```
$ gcc 1.c -g -o 1
```

```
$ gdb 1
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/1...done.
```

### Листинг 1.48: установим точку останова на printf()

```
(gdb) b printf
Breakpoint 1 at 0x80482f0
```

Запускаем. У нас нет исходного кода функции, поэтому **GDB!** не может его показать.

```
(gdb) run
Starting program: /home/dennis/polygon/1

Breakpoint 1, __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c:29
29      printf.c: No such file or directory.
```

Выдать 10 элементов стека. Левый столбец — это адрес в стеке.

```
(gdb) x/10w $esp
0xbffff11c: 0x0804844a    0x080484f0    0x00000001    0x00000002
0xbffff12c: 0x00000003    0x08048460    0x00000000
0xbffff13c: 0xb7e29905    0x00000001
```

Самый первый элемент это **RA!** (0x0804844a). Мы можем удостовериться в этом, дизассемблируя память по этому адресу:

<sup>67</sup> **GDB!**

```
(gdb) x/5i 0x0804844a
0x0804844a <main+45>: mov    $0x0,%eax
0x0804844f <main+50>: leave
0x08048450 <main+51>: ret
0x08048451:   xchg  %ax,%ax
0x08048453:   xchg  %ax,%ax
```

Две инструкции XCHG это холостые инструкции, аналогичные **NOP!**.

Второй элемент (0x080484f0) это адрес строки формата:

```
(gdb) x/s 0x080484f0
0x080484f0:   "a=%d; b=%d; c=%d"
```

Остальные 3 элемента (1, 2, 3) это аргументы функции printf(). Остальные элементы это может быть и мусор в стеке, но могут быть и значения от других функций, их локальные переменные, и т. д. Пока что мы можем игнорировать их.

Исполняем «finish». Это значит исполнять все инструкции до самого конца функции. В данном случае это означает исполнять до завершения printf().

```
(gdb) finish
Run till exit from #0  __printf (format=0x080484f0 "a=%d; b=%d; c=%d") at ↵
↳ printf.c:29
main () at 1.c:6
6          return 0;
Value returned is $2 = 13
```

**GDB!** показывает, что вернула printf() в EAX (13). Это, так же как и в примере с OllyDbg, количество напечатанных символов.

А ещё мы видим «return 0;» и что это выражение находится в файле 1.c в строке 6. Действительно, файл 1.c лежит в текущем директории и **GDB!** находит там эту строку. Как **GDB!** знает, какая строка Си-кода сейчас исполняется? Компилятор, генерируя отладочную информацию, также сохраняет информацию о соответствии строк в исходном коде и адресов инструкций. GDB это всё-таки отладчик уровня исходных текстов.

Посмотрим регистры. 13 в EAX:

```
(gdb) info registers
eax          0xd          13
ecx          0x0          0
edx          0x0          0
ebx          0xb7fc0000   -1208221696
esp          0xbffff120   0xbffff120
ebp          0xbffff138   0xbffff138
esi          0x0          0
edi          0x0          0
eip          0x0804844a   0x0804844a <main+45>
...
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Попробуем дизассемблировать текущие инструкции. Стрелка указывает на инструкцию, которая будет исполнена следующей.

```
(gdb) disas
Dump of assembler code for function main:
0x0804841d <+0>:    push   %ebp
0x0804841e <+1>:    mov    %esp,%ebp
0x08048420 <+3>:    and    $0xffffffff,%esp
0x08048423 <+6>:    sub    $0x10,%esp
0x08048426 <+9>:    movl   $0x3,0xc(%esp)
0x0804842e <+17>:   movl   $0x2,0x8(%esp)
0x08048436 <+25>:   movl   $0x1,0x4(%esp)
0x0804843e <+33>:   movl   $0x80484f0,(%esp)
0x08048445 <+40>:   call  0x80482f0 <printf@plt>
=> 0x0804844a <+45>:   mov    $0x0,%eax
0x0804844f <+50>:   leave
0x08048450 <+51>:   ret
End of assembler dump.
```

По умолчанию **GDB!** показывает дизассемблированный листинг в формате AT&T. Но можно также переключиться в формат Intel:

```
(gdb) set disassembly-flavor intel
(gdb) disas
Dump of assembler code for function main:
0x0804841d <+0>:    push   ebp
0x0804841e <+1>:    mov    ebp,esp
0x08048420 <+3>:    and    esp,0xffffffff
0x08048423 <+6>:    sub    esp,0x10
0x08048426 <+9>:    mov    DWORD PTR [esp+0xc],0x3
0x0804842e <+17>:   mov    DWORD PTR [esp+0x8],0x2
0x08048436 <+25>:   mov    DWORD PTR [esp+0x4],0x1
0x0804843e <+33>:   mov    DWORD PTR [esp],0x80484f0
0x08048445 <+40>:   call  0x80482f0 <printf@plt>
=> 0x0804844a <+45>:   mov    eax,0x0
0x0804844f <+50>:   leave
0x08048450 <+51>:   ret
End of assembler dump.
```

Исполняем следующую строку Си/Си++-кода. **GDB!** покажет закрывающуюся скобку, означая, что это конец блока в функции.

```
(gdb) step
7      };
```

Посмотрим регистры после исполнения инструкции MOV EAX, 0. EAX здесь уже действительно ноль.

```
(gdb) info registers
eax            0x0          0
ecx            0x0          0
edx            0x0          0
ebx            0xb7fc0000   -1208221696
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



esp	0xbffff120	0xbffff120
ebp	0xbffff138	0xbffff138
esi	0x0	0
edi	0x0	0
eip	0x804844f	0x804844f <main+50>
...		

### x64: 8 целочисленных аргументов

Для того чтобы посмотреть, как остальные аргументы будут передаваться через стек, изменим пример ещё раз, увеличив количество передаваемых аргументов до 9 (строка формата printf() и 8 переменных типа *int*):

```
#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4,
    ↵ 4, 5, 6, 7, 8);
    return 0;
};
```

### MSVC

Как уже было сказано ранее, первые 4 аргумента в Win64 передаются в регистрах RCX, RDX, R8, R9, а остальные — через стек. Здесь мы это и видим. Впрочем, инструкция PUSH не используется, вместо неё при помощи MOV значения сразу записываются в стек.

Листинг 1.49: MSVC 2012 x64

```
$SG2923 DB      'a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d', 0aH, 00H

main    PROC
sub     rsp, 88

        mov     DWORD PTR [rsp+64], 8
        mov     DWORD PTR [rsp+56], 7
        mov     DWORD PTR [rsp+48], 6
        mov     DWORD PTR [rsp+40], 5
        mov     DWORD PTR [rsp+32], 4
        mov     r9d, 3
        mov     r8d, 2
        mov     edx, 1
        lea    rcx, OFFSET FLAT:$SG2923
        call   printf

        ; возврат 0
        xor     eax, eax

        add     rsp, 88
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        ret     0
main    ENDP
_TEXT  ENDS
END

```

Наблюдательный читатель может спросить, почему для значений типа *int* отводится 8 байт, ведь нужно только 4? Да, это нужно запомнить: для значений всех типов более коротких чем 64-бита, отводится 8 байт. Это сделано для удобства: так всегда легко рассчитать адрес того или иного аргумента. К тому же, все они расположены по выровненным адресам в памяти. В 32-битных средах точно также: для всех типов резервируется 4 байта в стеке.

## GCC

В \*NIX-системах для x86-64 ситуация похожая, вот только первые 6 аргументов передаются через RDI, RSI, RDX, RCX, R8, R9. Остальные — через стек. GCC генерирует код, записывающий указатель на строку в EDI вместо RDI — это мы уже рассмотрели чуть раньше: ?? (стр. ??).

Почему перед вызовом `printf()` очищается регистр EAX мы уже рассмотрели ранее ?? (стр. ??).

Листинг 1.50: Оптимизирующий GCC 4.4.6 x64

```

.LC0:
    .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"

main:
    sub     rsp, 40

    mov     r9d, 5
    mov     r8d, 4
    mov     ecx, 3
    mov     edx, 2
    mov     esi, 1
    mov     edi, OFFSET FLAT:.LC0
    xor     eax, eax ; количество переданных векторных регистров
    mov     DWORD PTR [rsp+16], 8
    mov     DWORD PTR [rsp+8], 7
    mov     DWORD PTR [rsp], 6
    call    printf

    ; возврат 0

    xor     eax, eax
    add     rsp, 40
    ret

```

## GCC + GDB

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Попробуем этот пример в **GDB!**.

```
$ gcc -g 2.c -o 2
```

```
$ gdb 2
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/2...done.
```

Листинг 1.51: ставим точку останова на `printf()`, запускаем

```
(gdb) b printf
Breakpoint 1 at 0x400410
(gdb) run
Starting program: /home/dennis/polygon/2

Breakpoint 1, __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d;
↳ ; g=%d; h=%d\n") at printf.c:29
29 printf.c: No such file or directory.
```

В регистрах RSI/RDX/RCX/R8/R9 всё предсказуемо. А RIP содержит адрес самой первой инструкции функции `printf()`.

```
(gdb) info registers
rax            0x0          0
rbx            0x0          0
rcx            0x3          3
rdx            0x2          2
rsi            0x1          1
rdi            0x400628 4195880
rbp            0x7fffffffdf60 0x7fffffffdf60
rsp            0x7fffffffdf38 0x7fffffffdf38
r8             0x4          4
r9             0x5          5
r10            0x7fffffffce0 140737488346336
r11            0x7ffff7a65f60 140737348263776
r12            0x400440 4195392
r13            0x7fffffff040 140737488347200
r14            0x0          0
r15            0x0          0
rip            0x7ffff7a65f60 0x7ffff7a65f60 <__printf>
...
```

Листинг 1.52: смотрим на строку формата

```
(gdb) x/s $rdi
0x400628: "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
```

Дампим стек на этот раз с командой `x/g` — *g* означает *giant words*, т.е. 64-битные слова.

```
(gdb) x/10g $rsp
0x7fffffffdf38: 0x0000000000400576      0x0000000000000006
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

0x7fffffffdf48: 0x0000000000000007	0x00007fff00000008
0x7fffffffdf58: 0x0000000000000000	0x0000000000000000
0x7fffffffdf68: 0x00007ffff7a33de5	0x0000000000000000
0x7fffffffdf78: 0x00007ffffffe048	0x0000000100000000

Самый первый элемент стека, как и в прошлый раз, это **RA!**. Через стек также передаются 3 значения: 6, 7, 8. Видно, что 8 передается с неочищенной старшей 32-битной частью: 0x00007fff00000008. Это нормально, ведь передаются числа типа *int*, а они 32-битные. Так что в старшей части регистра или памяти стека остался «случайный мусор».

**GDB!** показывает всю функцию `main()`, если попытаться посмотреть, куда вернется управление после исполнения `printf()`.

```
(gdb) set disassembly-flavor intel
(gdb) disas 0x000000000400576
Dump of assembler code for function main:
   0x00000000040052d <+0>:   push   rbp
   0x00000000040052e <+1>:   mov    rbp, rsp
   0x000000000400531 <+4>:   sub   rsp, 0x20
   0x000000000400535 <+8>:   mov   DWORD PTR [rsp+0x10], 0x8
   0x00000000040053d <+16>:  mov   DWORD PTR [rsp+0x8], 0x7
   0x000000000400545 <+24>:  mov   DWORD PTR [rsp], 0x6
   0x00000000040054c <+31>:  mov   r9d, 0x5
   0x000000000400552 <+37>:  mov   r8d, 0x4
   0x000000000400558 <+43>:  mov   ecx, 0x3
   0x00000000040055d <+48>:  mov   edx, 0x2
   0x000000000400562 <+53>:  mov   esi, 0x1
   0x000000000400567 <+58>:  mov   edi, 0x400628
   0x00000000040056c <+63>:  mov   eax, 0x0
   0x000000000400571 <+68>:  call  0x400410 <printf@plt>
   0x000000000400576 <+73>:  mov   eax, 0x0
   0x00000000040057b <+78>:  leave
   0x00000000040057c <+79>:  ret
End of assembler dump.
```

Заканчиваем исполнение `printf()`, исполняем инструкцию обнуляющую EAX, удостоверяемся что в регистре EAX именно ноль. RIP указывает сейчас на инструкцию LEAVE, т.е. предпоследнюю в функции `main()`.

```
(gdb) finish
Run till exit from #0  __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d; e=
↳ =%d; f=%d; g=%d; h=%d\n") at printf.c:29
a=1; b=2; c=3; d=4; e=5; f=6; g=7; h=8
main () at 2.c:6
6         return 0;
Value returned is $1 = 39
(gdb) next
7         };
(gdb) info registers
rax                0x0          0
rbx                0x0          0
rcx                0x26        38
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

rdx	0x7ffff7dd59f0	140737351866864
rsi	0x7fffffd9	2147483609
rdi	0x0 0	
rbp	0x7ffffffffffdf60	0x7ffffffffffdf60
rsp	0x7ffffffffffdf40	0x7ffffffffffdf40
r8	0x7ffff7dd26a0	140737351853728
r9	0x7ffff7a60134	140737348239668
r10	0x7ffffffffffd5b0	140737488344496
r11	0x7ffff7a95900	140737348458752
r12	0x400440 4195392	
r13	0x7ffffffffffe040	140737488347200
r14	0x0 0	
r15	0x0 0	
rip	0x40057b 0x40057b	<main+78>
...		

## 1.11.2. ARM

### ARM: 3 целочисленных аргумента

В ARM традиционно принята такая схема передачи аргументов в функцию: 4 первых аргумента через регистры R0-R3; а остальные — через стек. Это немного похоже на то, как аргументы передаются в `fastcall` (?? (стр. ??)) или `win64` (?? (стр. ??)).

### 32-битный ARM

#### Неоптимизирующий Keil 6/2013 (Режим ARM)

Листинг 1.53: Неоптимизирующий Keil 6/2013 (Режим ARM)

```
.text:00000000 main
.text:00000000 10 40 2D E9  STMFDP SP!, {R4,LR}
.text:00000004 03 30 A0 E3  MOV    R3, #3
.text:00000008 02 20 A0 E3  MOV    R2, #2
.text:0000000C 01 10 A0 E3  MOV    R1, #1
.text:00000010 08 00 8F E2  ADR    R0, aADBDCD ; "a=%d; b=%d; c=%d"
.text:00000014 06 00 00 EB  BL    __2printf
.text:00000018 00 00 A0 E3  MOV    R0, #0 ; return 0
.text:0000001C 10 80 BD E8  LDMFDP SP!, {R4,PC}
```

Итак, первые 4 аргумента передаются через регистры R0-R3, по порядку: указатель на формат-строку для `printf()` в R0, затем 1 в R1, 2 в R2 и 3 в R3.

Инструкция на 0x18 записывает 0 в R0 — это выражение в Си `return 0`. Пока что здесь нет ничего необычного. Оптимизирующий Keil 6/2013 генерирует точно такой же код.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

## Оптимизирующий Keil 6/2013 (Режим Thumb)

Листинг 1.54: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
.text:00000000 main
.text:00000000 10 B5          PUSH   {R4,LR}
.text:00000002 03 23          MOVVS  R3, #3
.text:00000004 02 22          MOVVS  R2, #2
.text:00000006 01 21          MOVVS  R1, #1
.text:00000008 02 A0          ADR    R0, aADBDCD      ; "a=%d; b=%d; c=%d"
.text:0000000A 00 F0 0D F8      BL     __2printf
.text:0000000E 00 20          MOVVS  R0, #0
.text:00000010 10 BD          POP    {R4,PC}
```

Здесь нет особых отличий от неоптимизированного варианта для режима ARM.

## Оптимизирующий Keil 6/2013 (Режим ARM) + убираем return

Немного переделаем пример, убрав *return 0*:

```
#include <stdio.h>

void main()
{
    printf("a=%d; b=%d; c=%d", 1, 2, 3);
};
```

Результат получится необычным:

Листинг 1.55: Оптимизирующий Keil 6/2013 (Режим ARM)

```
.text:00000014 main
.text:00000014 03 30 A0 E3      MOV    R3, #3
.text:00000018 02 20 A0 E3      MOV    R2, #2
.text:0000001C 01 10 A0 E3      MOV    R1, #1
.text:00000020 1E 0E 8F E2      ADR    R0, aADBDCD      ; "a=%d; b=%d; c=%d\n"
.text:00000024 CB 18 00 EA      B     __2printf
```

Это оптимизированная версия (-O3) для режима ARM, и здесь мы видим последнюю инструкцию B вместо привычной нам BL. Отличия между этой оптимизированной версией и предыдущей, скомпилированной без оптимизации, ещё и в том, что здесь нет пролога и эпилога функции (инструкций, сохраняющих состояние регистров R0 и **LR!**). Инструкция B просто переходит на другой адрес, без манипуляций с регистром **LR!**, то есть это аналог JMP в x86. Почему это работает нормально? Потому что этот код эквивалентен предыдущему.

Основных причин две: 1) стек не модифицируется, как и [указатель стека SPI](#); 2) вызов функции printf() последний, после него ничего не происходит. Функция printf(), отработав, просто возвращает управление по адресу, записанному в **LR!**.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Но в **LR!** находится адрес места, откуда была вызвана наша функция! А следовательно, управление из `printf()` вернется сразу туда.

Значит нет нужды сохранять **LR!**, потому что нет нужды модифицировать **LR!**. А нет нужды модифицировать **LR!**, потому что нет иных вызовов функций, кроме `printf()`, к тому же, после этого вызова не нужно ничего здесь больше делать! Поэтому такая оптимизация возможна.

Эта оптимизация часто используется в функциях, где последнее выражение — это вызов другой функции.

Ещё один похожий пример описан здесь: ?? (стр. ??).

## ARM64

### Неоптимизирующий GCC (Linaro) 4.9

Листинг 1.56: Неоптимизирующий GCC (Linaro) 4.9

```
.LC1:
    .string "a=%d; b=%d; c=%d"
f2:
; сохранить FP и LR в стековом фрейме:
    stp    x29, x30, [sp, -16]!
; установить стековый фрейм (FP=SP):
    add    x29, sp, 0
    adrp   x0, .LC1
    add    x0, x0, :lo12:.LC1
    mov    w1, 1
    mov    w2, 2
    mov    w3, 3
    bl     printf
    mov    w0, 0
; восстановить FP и LR
    ldp    x29, x30, [sp], 16
    ret
```

Итак, первая инструкция `STP (Store Pair)` сохраняет **FP!** (X29) и **LR!** (X30) в стеке. Вторая инструкция `ADD X29, SP, 0` формирует стековый фрейм. Это просто запись значения **SP!** в X29.

Далее уже знакомая пара инструкций `ADRP/ADD` формирует указатель на строку.

`lo12` означает младшие 12 бит, т.е., линкер запишет младшие 12 бит адреса метки `LC1` в опкод инструкции `ADD`. 1, 2 и 3 это 32-битные значения типа `int`, так что они загружаются в 32-битные части регистров <sup>68</sup>

Оптимизирующий GCC (Linaro) 4.9 генерирует почти такой же код.

<sup>68</sup>Изменение 1 на 1L изменит эту константу на 64-битную и она будет загружаться в 64-битный регистр. См. о целочисленных константах/литералах: 1, 2.

## ARM: 8 целочисленных аргументов

Снова воспользуемся примером с 9-ю аргументами из предыдущей секции: ?? (стр. ??).

```
#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4,
    4, 5, 6, 7, 8);
    return 0;
};
```

## Оптимизирующий Keil 6/2013: Режим ARM

```
.text:00000028          main
.text:00000028
.text:00000028          var_18 = -0x18
.text:00000028          var_14 = -0x14
.text:00000028          var_4  = -4
.text:00000028
.text:00000028 04 E0 2D E5  STR    LR, [SP,#var_4]!
.text:0000002C 14 D0 4D E2  SUB    SP, SP, #0x14
.text:00000030 08 30 A0 E3  MOV    R3, #8
.text:00000034 07 20 A0 E3  MOV    R2, #7
.text:00000038 06 10 A0 E3  MOV    R1, #6
.text:0000003C 05 00 A0 E3  MOV    R0, #5
.text:00000040 04 C0 8D E2  ADD    R12, SP, #0x18+var_14
.text:00000044 0F 00 8C E8  STMIA  R12, {R0-R3}
.text:00000048 04 00 A0 E3  MOV    R0, #4
.text:0000004C 00 00 8D E5  STR    R0, [SP,#0x18+var_18]
.text:00000050 03 30 A0 E3  MOV    R3, #3
.text:00000054 02 20 A0 E3  MOV    R2, #2
.text:00000058 01 10 A0 E3  MOV    R1, #1
.text:0000005C 6E 0F 8F E2  ADR    R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d;
    d=%d; e=%d; f=%d; g=%"...
.text:00000060 BC 18 00 EB  BL    __2printf
.text:00000064 14 D0 8D E2  ADD    SP, SP, #0x14
.text:00000068 04 F0 9D E4  LDR    PC, [SP+4+var_4],#4
```

Этот код можно условно разделить на несколько частей:

- Пролог функции:

Самая первая инструкция `STR LR, [SP,#var_4]!` сохраняет в стеке **LR!**, ведь нам придется использовать этот регистр для вызова `printf()`. Восклицательный знак в конце означает *pre-index*. Это значит, что в начале **SP!** должно быть уменьшено на 4, затем по адресу в **SP!** должно быть записано значение **LR!**.

Это аналог знакомой в x86 инструкции `PUSH`. Читайте больше об этом: ?? (стр. ??).

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



Вторая инструкция `SUB SP, SP, #0x14` уменьшает **указатель стека SP!**, но, на самом деле, эта процедура нужна для выделения в локальном стеке места размером `0x14` (20) байт. Действительно, нам нужно передать 5 32-битных значений через стек в `printf()`. Каждое значение занимает 4 байта, все вместе —  $5 * 4 = 20$ . Остальные 4 32-битных значения будут переданы через регистры.

- Передача 5, 6, 7 и 8 через стек: они записываются в регистры R0, R1, R2 и R3 соответственно.

Затем инструкция `ADD R12, SP, #0x18+var_14` записывает в регистр R12 адрес места в стеке, куда будут помещены эти 4 значения. `var_14` — это макрос ассемблера, равный `-0x14`. Такие макросы создает **IDA!**, чтобы удобнее было показывать, как код обращается к стеку.

Макросы `var_?`, создаваемые **IDA!**, отражают локальные переменные в стеке. Так что в R12 будет записано `SP+4`.

Следующая инструкция `STMIA R12, R0-R3` записывает содержимое регистров R0-R3 по адресу в памяти, на который указывает R12.

Инструкция `STMIA` означает *Store Multiple Increment After*.

*Increment After* означает, что R12 будет увеличиваться на 4 после записи каждого значения регистра.

- Передача 4 через стек: 4 записывается в R0, затем инструкция `STR R0, [SP, #0x18+var_18]` записывает его в стек. `var_18` равен `-0x18`, смещение будет 0, так что значение из регистра R0 (4) запишется туда, куда указывает **SP!**.
- Передача 1, 2 и 3 через регистры:

Значения для первых трех чисел (a, b, c) (1, 2, 3 соответственно) передаются в регистрах R1, R2 и R3 перед самым вызовом `printf()`.

- Вызов `printf()`.

- Эпилог функции:

Инструкция `ADD SP, SP, #0x14` возвращает **SP!** на прежнее место, аннулируя таким образом всё, что было записано в стек. Конечно, то что было записано в стек, там пока и останется, но всё это будет многократно перезаписано во время исполнения последующих функций.

Инструкция `LDR PC, [SP+4+var_4], #4` загружает в **PC!** сохраненное значение **LR!** из стека, обеспечивая таким образом выход из функции.

Здесь нет восклицательного знака — действительно, сначала **PC!** загружается из места, куда указывает **SP!** ( $4 + var_4 = 4 + (-4) = 0$ ), так что эта инструкция аналогична `LDR PC, [SP], #4`), затем **SP!** увеличивается на 4. Это называется *post-index*<sup>69</sup>. Почему **IDA!** показывает инструкцию именно так? Потому что она хочет показать разметку стека и тот факт, что `var_4` выделена в локальном стеке именно для сохраненного значения **LR!**. Эта инструкция в каком-то смысле аналогична `POP PC` в x86<sup>70</sup>.

<sup>69</sup>Читайте больше об этом: **??** (стр. **??**).

<sup>70</sup>В x86 невозможно установить значение IP/EIP/RIP используя `POP`, но будем надеяться, вы

## Оптимизирующий Keil 6/2013: Режим Thumb

```
.text:0000001C      printf_main2
.text:0000001C
.text:0000001C      var_18 = -0x18
.text:0000001C      var_14 = -0x14
.text:0000001C      var_8  = -8
.text:0000001C
.text:0000001C 00 B5      PUSH    {LR}
.text:0000001E 08 23      MOVS    R3, #8
.text:00000020 85 B0      SUB     SP, SP, #0x14
.text:00000022 04 93      STR     R3, [SP,#0x18+var_8]
.text:00000024 07 22      MOVS    R2, #7
.text:00000026 06 21      MOVS    R1, #6
.text:00000028 05 20      MOVS    R0, #5
.text:0000002A 01 AB      ADD     R3, SP, #0x18+var_14
.text:0000002C 07 C3      STMIA  R3!, {R0-R2}
.text:0000002E 04 20      MOVS    R0, #4
.text:00000030 00 90      STR     R0, [SP,#0x18+var_18]
.text:00000032 03 23      MOVS    R3, #3
.text:00000034 02 22      MOVS    R2, #2
.text:00000036 01 21      MOVS    R1, #1
.text:00000038 A0 A0      ADR     R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d;
    d=%d; e=%d; f=%d; g=%"...
.text:0000003A 06 F0 D9 F8 BL      __2printf
.text:0000003E
.text:0000003E      loc_3E ; CODE XREF: example13_f+16
.text:0000003E 05 B0      ADD     SP, SP, #0x14
.text:00000040 00 BD      POP     {PC}
```

Это почти то же самое что и в предыдущем примере, только код для Thumb и значения помещаются в стек немного иначе: сначала 8 за первый раз, затем 5, 6, 7 за второй раз и 4 за третий раз.

## Оптимизирующий Xcode 4.6.3 (LLVM): Режим ARM

```
__text:0000290C      _printf_main2
__text:0000290C
__text:0000290C      var_1C = -0x1C
__text:0000290C      var_C  = -0xC
__text:0000290C
__text:0000290C 80 40 2D E9      STMFD  SP!, {R7,LR}
__text:00002910 0D 70 A0 E1      MOV    R7, SP
__text:00002914 14 D0 4D E2      SUB    SP, SP, #0x14
__text:00002918 70 05 01 E3      MOV    R0, #0x1570
__text:0000291C 07 C0 A0 E3      MOV    R12, #7
__text:00002920 00 00 40 E3      MOVT  R0, #0
__text:00002924 04 20 A0 E3      MOV    R2, #4
__text:00002928 00 00 8F E0      ADD    R0, PC, R0
```

поняли аналогию.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

__text:0000292C 06 30 A0 E3  MOV    R3, #6
__text:00002930 05 10 A0 E3  MOV    R1, #5
__text:00002934 00 20 8D E5  STR    R2, [SP,#0x1C+var_1C]
__text:00002938 0A 10 8D E9  STMFA  SP, {R1,R3,R12}
__text:0000293C 08 90 A0 E3  MOV    R9, #8
__text:00002940 01 10 A0 E3  MOV    R1, #1
__text:00002944 02 20 A0 E3  MOV    R2, #2
__text:00002948 03 30 A0 E3  MOV    R3, #3
__text:0000294C 10 90 8D E5  STR    R9, [SP,#0x1C+var_C]
__text:00002950 A4 05 00 EB  BL    _printf
__text:00002954 07 D0 A0 E1  MOV    SP, R7
__text:00002958 80 80 BD E8  LDMFD SP!, {R7,PC}

```

Почти то же самое, что мы уже видели, за исключением того, что STMFA (Store Multiple Full Ascending) — это синоним инструкции STMIB (Store Multiple Increment Before). Эта инструкция увеличивает **SP!** и только затем записывает в память значение очередного регистра, но не наоборот.

Далее бросается в глаза то, что инструкции как будто бы расположены случайно. Например, значение в регистре R0 подготавливается в трех местах, по адресам 0x2918, 0x2920 и 0x2928, когда это можно было бы сделать в одном месте. Однако, у оптимизирующего компилятора могут быть свои доводы о том, как лучше составлять инструкции друг с другом для лучшей эффективности исполнения. Процессор обычно пытается исполнять одновременно идущие друг за другом инструкции. К примеру, инструкции MOV R0, #0 и ADD R0, PC, R0 не могут быть исполнены одновременно, потому что обе инструкции модифицируют регистр R0. А вот инструкции MOV R0, #0 и MOV R2, #4 легко можно исполнить одновременно, потому что эффекты от их исполнения никак не конфликтуют друг с другом. Вероятно, компилятор старается генерировать код именно таким образом там, где это возможно.

### Оптимизирующий Xcode 4.6.3 (LLVM): Режим Thumb-2

```

__text:00002BA0          _printf_main2
__text:00002BA0
__text:00002BA0          var_1C = -0x1C
__text:00002BA0          var_18 = -0x18
__text:00002BA0          var_C  = -0xC
__text:00002BA0
__text:00002BA0 80 B5          PUSH    {R7,LR}
__text:00002BA2 6F 46          MOV     R7, SP
__text:00002BA4 85 B0          SUB     SP, SP, #0x14
__text:00002BA6 41 F2 D8 20    MOVW   R0, #0x12D8
__text:00002BAA 4F F0 07 0C    MOV.W  R12, #7
__text:00002BAE C0 F2 00 00    MOVT.W R0, #0
__text:00002BB2 04 22          MOVS   R2, #4
__text:00002BB4 78 44          ADD    R0, PC ; char *
__text:00002BB6 06 23          MOVS   R3, #6
__text:00002BB8 05 21          MOVS   R1, #5
__text:00002BBA 0D F1 04 0E    ADD.W  LR, SP, #0x1C+var_18
__text:00002BBE 00 92          STR    R2, [SP,#0x1C+var_1C]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

__text:00002BC0 4F F0 08 09    MOV.W    R9, #8
__text:00002BC4 8E E8 0A 10    STMIA.W LR, {R1,R3,R12}
__text:00002BC8 01 21         MOVS    R1, #1
__text:00002BCA 02 22         MOVS    R2, #2
__text:00002BCC 03 23         MOVS    R3, #3
__text:00002BCE CD F8 10 90    STR.W   R9, [SP,#0x1C+var_C]
__text:00002BD2 01 F0 0A EA    BLX    _printf
__text:00002BD6 05 B0         ADD     SP, SP, #0x14
__text:00002BD8 80 BD         POP     {R7,PC}

```

Почти то же самое, что и в предыдущем примере, лишь за тем исключением, что здесь используются Thumb/Thumb-2-инструкции.

## ARM64

### Неоптимизирующий GCC (Linaro) 4.9

Листинг 1.57: Неоптимизирующий GCC (Linaro) 4.9

```

.LC2:
.string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
f3:
; выделить больше места в стеке:
sub    sp, sp, #32
; сохранить FP и LR в стековом фрейме:
stp    x29, x30, [sp,16]
; установить указатель фрейма (FP=SP+16):
add    x29, sp, 16
adrp   x0, .LC2 ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
add    x0, x0, :lo12:.LC2
mov    w1, 8           ; 9-й аргумент
str    w1, [sp]       ; сохранить 9-й аргумент в стеке
mov    w1, 1
mov    w2, 2
mov    w3, 3
mov    w4, 4
mov    w5, 5
mov    w6, 6
mov    w7, 7
bl     printf
sub    sp, x29, #16
; восстановить FP и LR
ldp   x29, x30, [sp,16]
add    sp, sp, 32
ret

```

Первые 8 аргументов передаются в X- или W-регистрах: [*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)]<sup>71</sup>. Указатель на строку требует

<sup>71</sup>Также доступно здесь: [http://infocenter.arm.com/help/topic/com.arm.doc.ih0055b/IHI0055B\\_aapcs64.pdf](http://infocenter.arm.com/help/topic/com.arm.doc.ih0055b/IHI0055B_aapcs64.pdf)

64-битного регистра, так что он передается в X0. Все остальные значения имеют 32-битный тип *int*, так что они записываются в 32-битные части регистров (W-). Девятый аргумент (8) передается через стек. Действительно, невозможно передать большое количество аргументов в регистрах, потому что количество регистров ограничено.

Оптимизирующий GCC (Linaro) 4.9 генерирует почти такой же код.

### 1.11.3. MIPS

#### 3 целочисленных аргумента

##### Оптимизирующий GCC 4.4.5

Главное отличие от примера «Hello, world!» в том, что здесь на самом деле вызывается `printf()` вместо `puts()` и ещё три аргумента передаются в регистрах \$5...\$7 (или \$A0...\$A2). Вот почему эти регистры имеют префикс A-. Это значит, что они используются для передачи аргументов.

Листинг 1.58: Оптимизирующий GCC 4.4.5 (вывод на ассемблере)

```
$LC0:
    .ascii "a=%d; b=%d; c=%d\000"
main:
; пролог функции:
    lui    $28,%hi(__gnu_local_gp)
    addiu  $sp,$sp,-32
    addiu  $28,$28,%lo(__gnu_local_gp)
    sw     $31,28($sp)
; загрузить адрес printf():
    lw     $25,%call16(printf)($28)
; загрузить адрес текстовой строки и установить первый аргумент printf():
    lui    $4,%hi($LC0)
    addiu  $4,$4,%lo($LC0)
; установить второй аргумент printf():
    li     $5,1                # 0x1
; установить третий аргумент printf():
    li     $6,2                # 0x2
; вызов printf():
    jalr   $25
; установить четвертый аргумент printf() (branch delay slot):
    li     $7,3                # 0x3

; эпилог функции:
    lw     $31,28($sp)
; установить возвращаемое значение в 0:
    move   $2,$0
; возврат
    j      $31
    addiu  $sp,$sp,32 ; branch delay slot
```

Листинг 1.59: Оптимизирующий GCC 4.4.5 (IDA)

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:00000000 main:
.text:00000000
.text:00000000 var_10      = -0x10
.text:00000000 var_4      = -4
.text:00000000
; пролог функции:
.text:00000000          lui    $gp, (__gnu_local_gp >> 16)
.text:00000004          addiu  $sp, -0x20
.text:00000008          la     $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C          sw     $ra, 0x20+var_4($sp)
.text:00000010          sw     $gp, 0x20+var_10($sp)
; загрузить адрес printf():
.text:00000014          lw     $t9, (printf & 0xFFFF)($gp)
; загрузить адрес текстовой строки и установить первый аргумент printf():
.text:00000018          la     $a0, $LC0          # "a=%d; b=%d; c=%d"
; установить второй аргумент printf():
.text:00000020          li     $a1, 1
; установить третий аргумент printf():
.text:00000024          li     $a2, 2
; вызов printf():
.text:00000028          jalr  $t9
; установить четвертый аргумент printf() (branch delay slot):
.text:0000002C          li     $a3, 3
; эпилог функции:
.text:00000030          lw     $ra, 0x20+var_4($sp)
; установить возвращаемое значение в 0:
.text:00000034          move  $v0, $zero
; возврат
.text:00000038          jr     $ra
.text:0000003C          addiu  $sp, 0x20 ; branch delay slot

```

**IDA!** объединила пару инструкций LUI и ADDIU в одну псевдоинструкцию LA. Вот почему здесь нет инструкции по адресу 0x1C: потому что LA занимает 8 байт.

### Неоптимизирующий GCC 4.4.5

Неоптимизирующий GCC более многословен:

Листинг 1.60: Неоптимизирующий GCC 4.4.5 (вывод на ассемблере)

```

$LC0:
    .ascii "a=%d; b=%d; c=%d\000"
main:
; пролог функции:
    addiu  $sp,$sp,-32
    sw     $31,28($sp)
    sw     $fp,24($sp)
    move   $fp,$sp
    lui    $28,%hi(__gnu_local_gp)
    addiu  $28,$28,%lo(__gnu_local_gp)
; загрузить адрес текстовой строки:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        lui    $2,%hi($LC0)
        addiu  $2,$2,%lo($LC0)
; установить первый аргумент printf():
        move   $4,$2
; установить второй аргумент printf():
        li     $5,1           # 0x1
; установить третий аргумент printf():
        li     $6,2           # 0x2
; установить четвертый аргумент printf():
        li     $7,3           # 0x3
; получить адрес printf():
        lw     $2,%call16(printf)($28)
        nop
; вызов printf():
        move   $25,$2
        jalr   $25
        nop

; эпилог функции:
        lw     $28,16($fp)
; установить возвращаемое значение в 0:
        move   $2,$0
        move   $sp,$fp
        lw     $31,28($sp)
        lw     $fp,24($sp)
        addiu  $sp,$sp,32
; возврат
        j     $31
        nop

```

Листинг 1.61: Неоптимизирующий GCC 4.4.5 (IDA)

```

.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_8          = -8
.text:00000000 var_4          = -4
.text:00000000
; пролог функции:
.text:00000000          addiu   $sp, -0x20
.text:00000004          sw     $ra, 0x20+var_4($sp)
.text:00000008          sw     $fp, 0x20+var_8($sp)
.text:0000000C          move   $fp, $sp
.text:00000010          la     $gp, __gnu_local_gp
.text:00000018          sw     $gp, 0x20+var_10($sp)
; загрузить адрес текстовой строки:
.text:0000001C          la     $v0, aADBDCD      # "a=%d; b=%d; c=%d"
; установить первый аргумент printf():
.text:00000024          move   $a0, $v0
; установить второй аргумент printf():
.text:00000028          li     $a1, 1
; установить третий аргумент printf():
.text:0000002C          li     $a2, 2
; установить четвертый аргумент printf():

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:00000030      li      $a3, 3
; получить адрес printf():
.text:00000034      lw      $v0, (printf & 0xFFFF)($gp)
.text:00000038      or      $at, $zero
; вызов printf():
.text:0000003C      move   $t9, $v0
.text:00000040      jalr   $t9
.text:00000044      or      $at, $zero ; NOP
; эпилог функции:
.text:00000048      lw      $gp, 0x20+var_10($fp)
; установить возвращаемое значение в 0:
.text:0000004C      move   $v0, $zero
.text:00000050      move   $sp, $fp
.text:00000054      lw     $ra, 0x20+var_4($sp)
.text:00000058      lw     $fp, 0x20+var_8($sp)
.text:0000005C      addiu  $sp, 0x20
; возврат
.text:00000060      jr     $ra
.text:00000064      or     $at, $zero ; NOP

```

## 8 целочисленных аргументов

Снова воспользуемся примером с 9-ю аргументами из предыдущей секции: ?? (стр. ??).

```

#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4,
    ↵ 4, 5, 6, 7, 8);
    return 0;
};

```

### Оптимизирующий GCC 4.4.5

Только 4 первых аргумента передаются в регистрах \$A0 ...\$A3, так что остальные передаются через стек.

Это соглашение о вызовах O32 (самое популярное в мире MIPS). Другие соглашения о вызовах, или вручную написанный код на ассемблере, могут наделять регистры другими функциями.

SW означает «Store Word» (записать слово из регистра в память). В MIPS нет инструкции для записи значения в память, так что для этого используется пара инструкций (LI/SW).

Листинг 1.62: Оптимизирующий GCC 4.4.5 (вывод на ассемблере)

```

$LC0:
    .ascii "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

main:
; пролог функции:
    lui    $28,%hi(__gnu_local_gp)
    addiu  $sp,$sp,-56
    addiu  $28,$28,%lo(__gnu_local_gp)
    sw     $31,52($sp)
; передать 5-й аргумент в стеке:
    li     $2,4                # 0x4
    sw     $2,16($sp)
; передать 6-й аргумент в стеке:
    li     $2,5                # 0x5
    sw     $2,20($sp)
; передать 7-й аргумент в стеке:
    li     $2,6                # 0x6
    sw     $2,24($sp)
; передать 8-й аргумент в стеке:
    li     $2,7                # 0x7
    lw     $25,%call16(printf)($28)
    sw     $2,28($sp)
; передать 1-й аргумент в $a0:
    lui    $4,%hi($LC0)
; передать 9-й аргумент в стеке:
    li     $2,8                # 0x8
    sw     $2,32($sp)
    addiu  $4,$4,%lo($LC0)
; передать 2-й аргумент в $a1:
    li     $5,1                # 0x1
; передать 3-й аргумент в $a2:
    li     $6,2                # 0x2
; вызов printf():
    jalr   $25
; передать 4-й аргумент в $a3 (branch delay slot):
    li     $7,3                # 0x3

; эпилог функции:
    lw     $31,52($sp)
; установить возвращаемое значение в 0:
    move   $2,$0
; возврат
    j      $31
    addiu  $sp,$sp,56 ; branch delay slot

```

Листинг 1.63: Оптимизирующий GCC 4.4.5 (IDA)

```

.text:00000000 main:
.text:00000000
.text:00000000 var_28          = -0x28
.text:00000000 var_24          = -0x24
.text:00000000 var_20          = -0x20
.text:00000000 var_1c          = -0x1c
.text:00000000 var_18          = -0x18
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
.text:00000000

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; пролог функции:
.text:00000000      lui      $gp, (__gnu_local_gp >> 16)
.text:00000004      addiu   $sp, -0x38
.text:00000008      la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C      sw      $ra, 0x38+var_4($sp)
.text:00000010      sw      $gp, 0x38+var_10($sp)
; передать 5-й аргумент в стеке:
.text:00000014      li      $v0, 4
.text:00000018      sw      $v0, 0x38+var_28($sp)
; передать 6-й аргумент в стеке:
.text:0000001C      li      $v0, 5
.text:00000020      sw      $v0, 0x38+var_24($sp)
; передать 7-й аргумент в стеке:
.text:00000024      li      $v0, 6
.text:00000028      sw      $v0, 0x38+var_20($sp)
; передать 8-й аргумент в стеке:
.text:0000002C      li      $v0, 7
.text:00000030      lw      $t9, (printf & 0xFFFF)($gp)
.text:00000034      sw      $v0, 0x38+var_1C($sp)
; готовить 1-й аргумент в $a0:
.text:00000038      lui      $a0, ($LC0 >> 16) # "a=%d; b=%d;
      c=%d; d=%d; e=%d; f=%d; g=%"...
; передать 9-й аргумент в стеке:
.text:0000003C      li      $v0, 8
.text:00000040      sw      $v0, 0x38+var_18($sp)
; передать 1-й аргумент в $a0:
.text:00000044      la      $a0, ($LC0 & 0xFFFF) # "a=%d; b=%d;
      c=%d; d=%d; e=%d; f=%d; g=%"...
; передать 2-й аргумент в $a1:
.text:00000048      li      $a1, 1
; передать 3-й аргумент в $a2:
.text:0000004C      li      $a2, 2
; вызов printf():
.text:00000050      jalr   $t9
; передать 4-й аргумент в $a3 (branch delay slot):
.text:00000054      li      $a3, 3
; эпилог функции:
.text:00000058      lw      $ra, 0x38+var_4($sp)
; установить возвращаемое значение в 0:
.text:0000005C      move   $v0, $zero
; возврат
.text:00000060      jr     $ra
.text:00000064      addiu  $sp, 0x38 ; branch delay slot

```

## Неоптимизирующий GCC 4.4.5

Неоптимизирующий GCC более многословен:

Листинг 1.64: Неоптимизирующий GCC 4.4.5 (вывод на ассемблере)

```

$LC0:
      .ascii "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; пролог функции:
    addiu    $sp,$sp,-56
    sw      $31,52($sp)
    sw      $fp,48($sp)
    move    $fp,$sp
    lui     $28,%hi(__gnu_local_gp)
    addiu   $28,$28,%lo(__gnu_local_gp)
    lui     $2,%hi($LC0)
    addiu   $2,$2,%lo($LC0)
; передать 5-й аргумент в стеке:
    li      $3,4                # 0x4
    sw      $3,16($sp)
; передать 6-й аргумент в стеке:
    li      $3,5                # 0x5
    sw      $3,20($sp)
; передать 7-й аргумент в стеке:
    li      $3,6                # 0x6
    sw      $3,24($sp)
; передать 8-й аргумент в стеке:
    li      $3,7                # 0x7
    sw      $3,28($sp)
; передать 9-й аргумент в стеке:
    li      $3,8                # 0x8
    sw      $3,32($sp)
; передать 1-й аргумент в $a0:
    move    $4,$2
; передать 2-й аргумент в $a1:
    li      $5,1                # 0x1
; передать 3-й аргумент в $a2:
    li      $6,2                # 0x2
; передать 4-й аргумент в $a3:
    li      $7,3                # 0x3
; вызов printf():
    lw      $2,%call16(printf)($28)
    nop
    move    $25,$2
    jalr   $25
    nop
; эпилог функции:
    lw      $28,40($fp)
; установить возвращаемое значение в 0:
    move    $2,$0
    move    $sp,$fp
    lw      $31,52($sp)
    lw      $fp,48($sp)
    addiu   $sp,$sp,56
; возврат
    j      $31
    nop

```

Листинг 1.65: Неоптимизирующий GCC 4.4.5 (IDA)

```

.text:00000000 main:
.text:00000000

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:00000000 var_28      = -0x28
.text:00000000 var_24      = -0x24
.text:00000000 var_20      = -0x20
.text:00000000 var_1C      = -0x1C
.text:00000000 var_18      = -0x18
.text:00000000 var_10      = -0x10
.text:00000000 var_8       = -8
.text:00000000 var_4       = -4
.text:00000000
; пролог функции:
.text:00000000          addiu   $sp, -0x38
.text:00000004          sw     $ra, 0x38+var_4($sp)
.text:00000008          sw     $fp, 0x38+var_8($sp)
.text:0000000C          move  $fp, $sp
.text:00000010          la    $gp, __gnu_local_gp
.text:00000018          sw     $gp, 0x38+var_10($sp)
.text:0000001C          la    $v0, aADBCDDDEDFDGD # "a=%d; b=%d;
      c=%d; d=%d; e=%d; f=%d; g=%"...
; передать 5-й аргумент в стеке:
.text:00000024          li    $v1, 4
.text:00000028          sw     $v1, 0x38+var_28($sp)
; передать 6-й аргумент в стеке:
.text:0000002C          li    $v1, 5
.text:00000030          sw     $v1, 0x38+var_24($sp)
; передать 7-й аргумент в стеке:
.text:00000034          li    $v1, 6
.text:00000038          sw     $v1, 0x38+var_20($sp)
; передать 8-й аргумент в стеке:
.text:0000003C          li    $v1, 7
.text:00000040          sw     $v1, 0x38+var_1C($sp)
; передать 9-й аргумент в стеке:
.text:00000044          li    $v1, 8
.text:00000048          sw     $v1, 0x38+var_18($sp)
; передать 1-й аргумент в $a0:
.text:0000004C          move  $a0, $v0
; передать 2-й аргумент в $a1:
.text:00000050          li    $a1, 1
; передать 3-й аргумент в $a2:
.text:00000054          li    $a2, 2
; передать 4-й аргумент в $a3:
.text:00000058          li    $a3, 3
; вызов printf():
.text:0000005C          lw    $v0, (printf & 0xFFFF)($gp)
.text:00000060          or    $at, $zero
.text:00000064          move  $t9, $v0
.text:00000068          jalr  $t9
.text:0000006C          or    $at, $zero ; NOP
; эпилог функции:
.text:00000070          lw    $gp, 0x38+var_10($fp)
; установить возвращаемое значение в 0:
.text:00000074          move  $v0, $zero
.text:00000078          move  $sp, $fp
.text:0000007C          lw    $ra, 0x38+var_4($sp)
.text:00000080          lw    $fp, 0x38+var_8($sp)

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
.text:00000084      addiu   $sp, 0x38
; возврат
.text:00000088      jr      $ra
.text:0000008C      or      $at, $zero ; NOP
```

#### 1.11.4. Вывод

Вот примерный скелет вызова функции:

Листинг 1.66: x86

```
...
PUSH третий аргумент
PUSH второй аргумент
PUSH первый аргумент
CALL функция
; модифицировать указатель стека (если нужно)
```

Листинг 1.67: x64 (MSVC)

```
MOV RCX, первый аргумент
MOV RDX, второй аргумент
MOV R8, третий аргумент
MOV R9, 4-й аргумент
...
PUSH 5-й, 6-й аргумент, и т.д. (если нужно)
CALL функция
; модифицировать указатель стека (если нужно)
```

Листинг 1.68: x64 (GCC)

```
MOV RDI, первый аргумент
MOV RSI, второй аргумент
MOV RDX, третий аргумент
MOV RCX, 4-й аргумент
MOV R8, 5-й аргумент
MOV R9, 6-й аргумент
...
PUSH 7-й, 8-й аргумент, и т.д. (если нужно)
CALL функция
; модифицировать указатель стека (если нужно)
```

Листинг 1.69: ARM

```
MOV R0, первый аргумент
MOV R1, второй аргумент
MOV R2, третий аргумент
MOV R3, 4-й аргумент
; передать 5-й, 6-й аргумент, и т.д., в стеке (если нужно)
BL функция
; модифицировать указатель стека (если нужно)
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

## Листинг 1.70: ARM64

```

MOV X0, первый аргумент
MOV X1, второй аргумент
MOV X2, третий аргумент
MOV X3, 4-й аргумент
MOV X4, 5-й аргумент
MOV X5, 6-й аргумент
MOV X6, 7-й аргумент
MOV X7, 8-й аргумент
; передать 9-й, 10-й аргумент, и т.д., в стеке (если нужно)
BL функция
; модифицировать указатель стека (если нужно)

```

## Листинг 1.71: MIPS (соглашение о вызовах O32)

```

LI $4, первый аргумент ; AKA $A0
LI $5, второй аргумент ; AKA $A1
LI $6, третий аргумент ; AKA $A2
LI $7, 4-й аргумент ; AKA $A3
; передать 5-й, 6-й аргумент, и т.д., в стеке (если нужно)
LW temp_reg, адрес функции
JALR temp_reg

```

### 1.11.5. Кстати

Кстати, разница между способом передачи параметров принятая в x86, x64, fastcall, ARM и MIPS неплохо иллюстрирует тот важный момент, что процессору, в общем, всё равно, как будут передаваться параметры функций. Можно создать компилятор, который будет передавать их при помощи указателя на структуру с параметрами, не пользуясь стеком вообще.

Регистры \$A0...\$A3 в MIPS так названы только для удобства (это соглашение о вызовах O32). Программисты могут использовать любые другие регистры (может быть, только кроме \$ZERO) для передачи данных или любое другое соглашение о вызовах.

**CPU!** не знает о соглашениях о вызовах вообще.

Можно также вспомнить, что начинающие программисты на ассемблере передают параметры в другие функции обычно через регистры, без всякого явного порядка, или даже через глобальные переменные. И всё это нормально работает.

## 1.12. scanf()

Теперь попробуем использовать `scanf()`.

### 1.12.1. Простой пример

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
#include <stdio.h>

int main()
{
    int x;
    printf ("Enter X:\n");

    scanf ("%d", &x);

    printf ("You entered %d...\n", x);

    return 0;
};
```

Использовать `scanf()` в наши времена для того, чтобы спросить у пользователя что-то — не самая хорошая идея. Но так мы проиллюстрируем передачу указателя на переменную типа *int*.

### Об указателях

Это одна из фундаментальных вещей в программировании. Часто большой массив, структуру или объект передавать в другую функцию путем копирования данных невыгодно, а передать адрес массива, структуры или объекта куда проще. Например, если вы собираетесь вывести в консоль текстовую строку, достаточно только передать её адрес в ядро **OS!**

К тому же, если вызываемая функция (*callee*) должна изменить что-то в этом большом массиве или структуре, то возвращать её полностью так же абсурдно. Так что самое простое, что можно сделать, это передать в функцию-*callee* адрес массива или структуры, и пусть *callee* что-то там изменит.

Указатель в Си/Си++ — это просто адрес какого-либо места в памяти.

В x86 адрес представляется в виде 32-битного числа (т.е. занимает 4 байта), а в x86-64 как 64-битное число (занимает 8 байт). Кстати, отсюда негодование некоторых людей, связанное с переходом на x86-64 — на этой архитектуре все указатели занимают в 2 раза больше места, в том числе и в “дорогой” кэш-памяти.

При некотором упорстве можно работать только с безтиповыми указателями (*void\**), например, стандартная функция Си `memcpy()`, копирующая блок из одного места памяти в другое принимает на вход 2 указателя типа *void\**, потому что нельзя заранее предугадать, какого типа блок вы собираетесь копировать. Для копирования тип данных не важен, важен только размер блока.

Также указатели широко используются, когда функции нужно вернуть более одного значения (мы ещё вернемся к этому в будущем (**??** (стр. **??**))).

Функция `scanf()` — это как раз такой случай.

Помимо того, что этой функции нужно показать, сколько значений было прочитано успешно, ей ещё и нужно вернуть сами значения.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Тип указателя в Си/Си++ нужен только для проверки типов на стадии компиляции.

Внутри, в скомпилированном коде, никакой информации о типах указателей нет вообще.

## x86

### MSVC

Что получаем на ассемблере, компилируя в MSVC 2010:

```

CONST    SEGMENT
$SG3831  DB    'Enter X:', 0aH, 00H
$SG3832  DB    '%d', 00H
$SG3833  DB    'You entered %d...', 0aH, 00H
CONST    ENDS
PUBLIC   _main
EXTRN   _scanf:PROC
EXTRN   _printf:PROC
; Function compile flags: /OdtP
_TEXT   SEGMENT
_x$ = -4                                ; size = 4
_main   PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    push    OFFSET $SG3831 ; 'Enter X:'
    call    _printf
    add     esp, 4
    lea    eax, DWORD PTR _x$[ebp]
    push    eax
    push    OFFSET $SG3832 ; '%d'
    call    _scanf
    add     esp, 8
    mov     ecx, DWORD PTR _x$[ebp]
    push    ecx
    push    OFFSET $SG3833 ; 'You entered %d...'
    call    _printf
    add     esp, 8

    ; возврат 0
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main    ENDP
_TEXT   ENDS

```

Переменная x является локальной.

По стандарту Си/Си++ она доступна только из этой же функции и нигде более. Так получилось, что локальные переменные располагаются в стеке. Мо-

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



жет быть, можно было бы использовать и другие варианты, но в x86 это традиционно так.

Следующая после пролога инструкция `PUSH ECX` не ставит своей целью сохранить значение регистра `ECX`. (Заметьте отсутствие соответствующей инструкции `POP ECX` в конце функции).

Она на самом деле выделяет в стеке 4 байта для хранения `x` в будущем.

Доступ к `x` будет осуществляться при помощи объявленного макроса `_x$` (он равен `-4`) и регистра `EBP` указывающего на текущий фрейм.

Во всё время исполнения функции `EBP` указывает на текущий фрейм и через `EBP+смещение` можно получить доступ как к локальным переменным функции, так и аргументам функции.

Можно было бы использовать `ESP`, но он во время исполнения функции часто меняется, а это не удобно. Так что можно сказать, что `EBP` это замороженное состояние `ESP` на момент начала исполнения функции.

Разметка типичного стекового фрейма в 32-битной среде:

...	...
<code>EBP-8</code>	локальная переменная #2, маркируется в <b>IDA!</b> как <code>var_8</code>
<code>EBP-4</code>	локальная переменная #1, маркируется в <b>IDA!</b> как <code>var_4</code>
<code>EBP</code>	сохраненное значение <code>EBP</code>
<code>EBP+4</code>	адрес возврата
<code>EBP+8</code>	аргумент #1, маркируется в <b>IDA!</b> как <code>arg_0</code>
<code>EBP+0xC</code>	аргумент #2, маркируется в <b>IDA!</b> как <code>arg_4</code>
<code>EBP+0x10</code>	аргумент #3, маркируется в <b>IDA!</b> как <code>arg_8</code>
...	...

У функции `scanf()` в нашем примере два аргумента.

Первый — указатель на строку, содержащую `%d` и второй — адрес переменной `x`.

Вначале адрес `x` помещается в регистр `EAX` при помощи инструкции `lea eax, DWORD PTR _x$[ebp]`.

Инструкция `LEA` означает *load effective address*, и часто используется для формирования адреса чего-либо (?? (стр. ??)).

Можно сказать, что в данном случае `LEA` просто помещает в `EAX` результат суммы значения в регистре `EBP` и макроса `_x$`.

Это тоже что и `lea eax, [ebp-4]`.

Итак, от значения `EBP` отнимается 4 и помещается в `EAX`. Далее значение `EAX` заталкивается в стек и вызывается `scanf()`.

После этого вызывается `printf()`. Первый аргумент вызова строка: `You entered %d... \n`.

Второй аргумент: `mov ecx, [ebp-4]`. Эта инструкция помещает в `ECX` не адрес переменной `x`, а её значение.

---

Далее значение ECX заталкивается в стек и вызывается `printf()`.

## MSVC + OllyDbg

Попробуем этот же пример в OllyDbg. Загружаем, нажимаем F8 (сделать шаг, не входя в функцию) до тех пор, пока не окажемся в своем исполняемом файле, а не в `ntdll.dll`. Прокручиваем вверх до тех пор, пока не найдем `main()`. Щелкаем на первой инструкции (`PUSH EBP`), нажимаем F2 (*set a breakpoint*), затем F9 (*Run*) и точка останова срабатывает на начале `main()`.

Трассируем до того места, где готовится адрес переменной `x`:

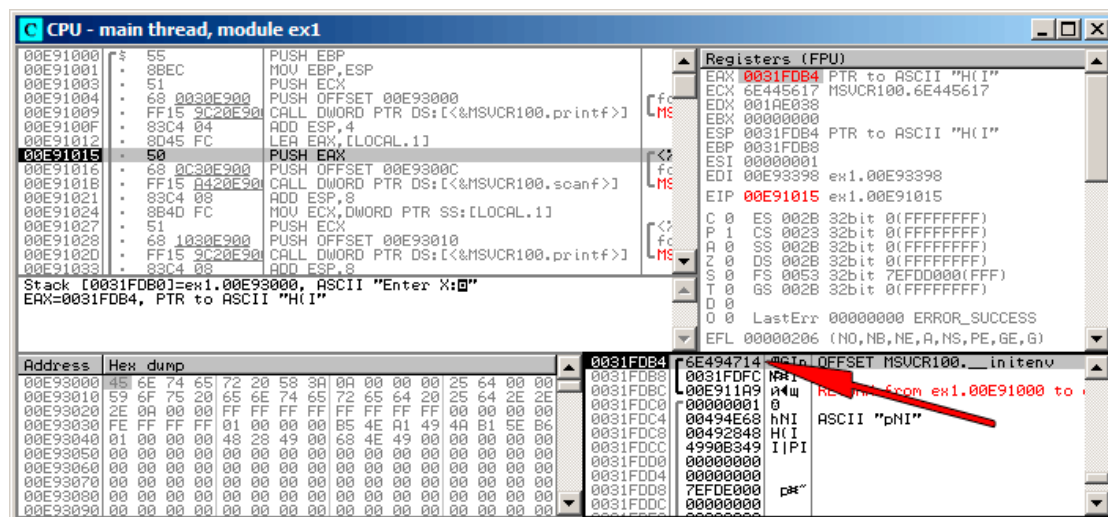


Рис. 1.13: OllyDbg: вычисляется адрес локальной переменной

На `EAX` в окне регистров можно нажать правой кнопкой и далее выбрать «Follow in stack». Этот адрес покажется в окне стека.

Смотрите, это переменная в локальном стеке. Там дорисована красная стрелка. И там сейчас какой-то мусор (`0x6E494714`). Адрес этого элемента стека сейчас, при помощи `PUSH` запишется в этот же стек рядом. Трассируем при помощи F8 вплоть до конца исполнения `scanf()`. А пока `scanf()` выполняется, в консольном окне, вводим, например, 123:

```
Enter X:
123
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Вот тут scanf() отработал:

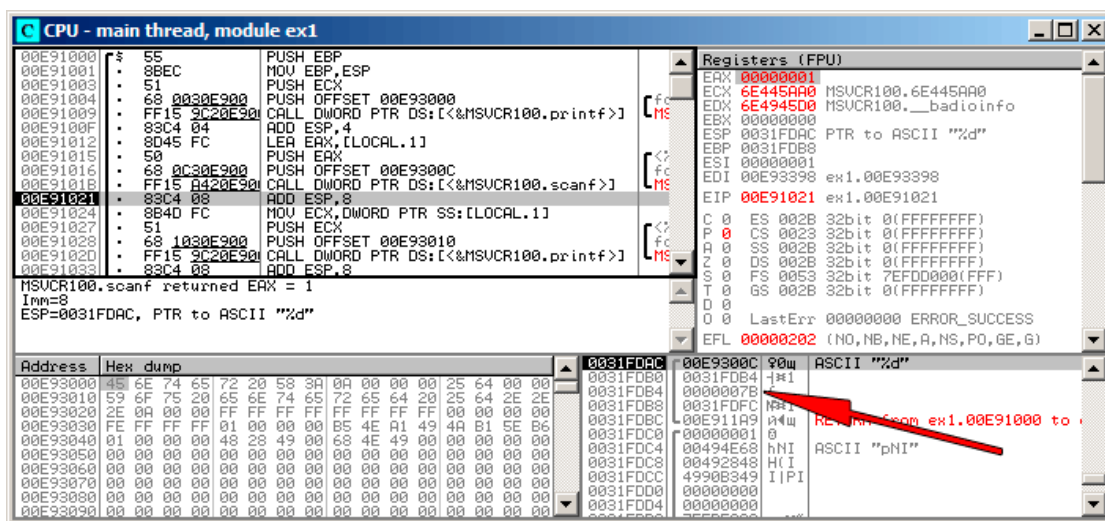


Рис. 1.14: OllyDbg: scanf() исполнялась

scanf() вернул 1 в EAX, что означает, что он успешно прочитал одно значение. В наблюдаемом нами элементе стека теперь 0x7B (123).

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Чуть позже это значение копируется из стека в регистр ECX и передается в printf():

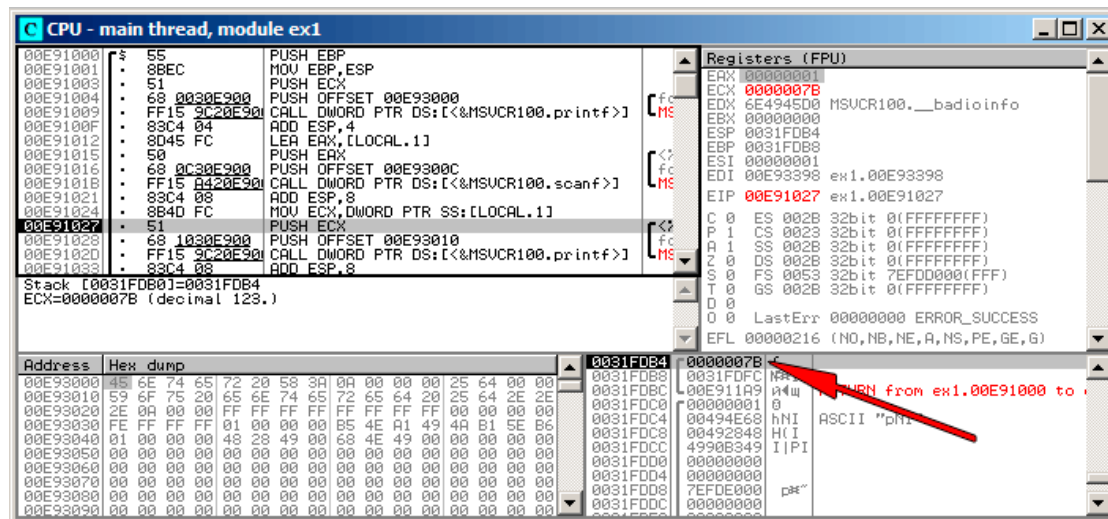


Рис. 1.15: OllyDbg: готовим значение для передачи в printf()

## GCC

Попробуем тоже самое скомпилировать в Linux при помощи GCC 4.4.1:

```
main                proc near

var_20              = dword ptr -20h
var_1C              = dword ptr -1Ch
var_4               = dword ptr -4

    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFF0h
    sub     esp, 20h
    mov     [esp+20h+var_20], offset aEnterX ; "Enter X:"
    call    _puts
    mov     eax, offset aD ; "%d"
    lea    edx, [esp+20h+var_4]
    mov     [esp+20h+var_1C], edx
    mov     [esp+20h+var_20], eax
    call    ___isoc99_scanf
    mov     edx, [esp+20h+var_4]
    mov     eax, offset aYouEnteredD___ ; "You entered %d...\n"
    mov     [esp+20h+var_1C], edx
    mov     [esp+20h+var_20], eax
    call    _printf
    mov     eax, 0
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

	leave
	retn
main	endp

GCC заменил первый вызов `printf()` на `puts()`. Почему это было сделано, уже было описано ранее (?? (стр. ??)).

Далее всё как и прежде — параметры заталкиваются через стек при помощи `MOV`.

## Кстати

Этот простой пример иллюстрирует то обстоятельство, что компилятор преобразует список выражений в Си/Си++-блоке просто в последовательный набор инструкций. Между выражениями в Си/Си++ ничего нет, и в итоговом машинном коде между ними тоже ничего нет, управление переходит от одной инструкции к следующей за ней.

## x64

Всё то же самое, только используются регистры вместо стека для передачи аргументов функций.

## MSVC

Листинг 1.72: MSVC 2012 x64

```

_DATA SEGMENT
$SG1289 DB 'Enter X:', 0aH, 00H
$SG1291 DB '%d', 00H
$SG1292 DB 'You entered %d...', 0aH, 00H
_DATA ENDS

_TEXT SEGMENT
x$ = 32
main PROC
$LN3:
    sub     rsp, 56
    lea    rcx, OFFSET FLAT:$SG1289 ; 'Enter X:'
    call   printf
    lea    rdx, QWORD PTR x$[rsp]
    lea    rcx, OFFSET FLAT:$SG1291 ; '%d'
    call   scanf
    mov    edx, DWORD PTR x$[rsp]
    lea    rcx, OFFSET FLAT:$SG1292 ; 'You entered %d...'
    call   printf

    ; возврат 0
    xor    eax, eax
    add    rsp, 56

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

main    ret     0
        ENDP
_TEXT  ENDS

```

## GCC

Листинг 1.73: Оптимизирующий GCC 4.4.6 x64

```

.LC0:
.string "Enter X:"
.LC1:
.string "%d"
.LC2:
.string "You entered %d...\n"
main:
    sub    rsp, 24
    mov    edi, OFFSET FLAT:.LC0 ; "Enter X:"
    call   puts
    lea   rsi, [rsp+12]
    mov    edi, OFFSET FLAT:.LC1 ; "%d"
    xor    eax, eax
    call  __isoc99_scanf
    mov    esi, DWORD PTR [rsp+12]
    mov    edi, OFFSET FLAT:.LC2 ; "You entered %d...\n"
    xor    eax, eax
    call  printf

    ; возврат 0
    xor    eax, eax
    add    rsp, 24
    ret

```

## ARM

### Оптимизирующий Keil 6/2013 (Режим Thumb)

```

.text:00000042          scanf_main
.text:00000042
.text:00000042          var_8          = -8
.text:00000042
.text:00000042 08 B5          PUSH    {R3,LR}
.text:00000044 A9 A0          ADR     R0, aEnterX ; "Enter X:\n"
.text:00000046 06 F0 D3 F8    BL     __2printf
.text:0000004A 69 46          MOV     R1, SP
.text:0000004C AA A0          ADR     R0, aD ; "%d"
.text:0000004E 06 F0 CD F8    BL     __0scanf
.text:00000052 00 99          LDR     R1, [SP,#8+var_8]
.text:00000054 A9 A0          ADR     R0, aYouEnteredD___ ; "You entered
%d...\n"

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
.text:00000056 06 F0 CB F8 BL    __2printf
.text:0000005A 00 20      MOVS   R0, #0
.text:0000005C 08 BD      POP   {R3,PC}
```

Чтобы `scanf()` мог вернуть значение, ему нужно передать указатель на переменную типа `int`. `int` — 32-битное значение, для его хранения нужно только 4 байта, и оно помещается в 32-битный регистр.

Место для локальной переменной `x` выделяется в стеке, **IDA!** наименовала её `var_8`. Впрочем, место для неё выделять не обязательно, т.к. **указатель стека SP!** уже указывает на место, свободное для использования. Так что значение указателя **SP!** копируется в регистр R1, и вместе с `format`-строкой, передается в `scanf()`.

Инструкции PUSH/POP в ARM работают иначе, чем в x86 (тут всё наоборот). Это синонимы инструкций STM/STMDB/LDM/LDMIA. И инструкция PUSH в начале записывает в стек значение, *затем* вычитает 4 из **SP!**. POP в начале прибавляет 4 к **SP!**, *затем* читает значение из стека. Так что после PUSH, **SP!** указывает на неиспользуемое место в стеке. Его и использует `scanf()`, а затем и `printf()`.

LDMIA означает *Load Multiple Registers Increment address After each transfer*. STMDB означает *Store Multiple Registers Decrement address Before each transfer*.

Позже, при помощи инструкции LDR, это значение перемещается из стека в регистр R1, чтобы быть переданным в `printf()`.

## ARM64

Листинг 1.74: Неоптимизирующий GCC 4.9.1 ARM64

```
1  .LC0:
2      .string "Enter X:"
3  .LC1:
4      .string "%d"
5  .LC2:
6      .string "You entered %d...\n"
7  scanf_main:
8      ; вычесть 32 из SP, затем сохранить FP и LR в стековом фрейме:
9      stp    x29, x30, [sp, -32]!
10     ; установить стековый фрейм (FP=SP)
11     add    x29, sp, 0
12     ; загрузить указатель на строку "Enter X:"
13     adrp   x0, .LC0
14     add    x0, x0, :lo12:.LC0
15     ; X0=указатель на строку "Enter X:"
16     ; вывести её:
17     bl    puts
18     ; загрузить указатель на строку "%d":
19     adrp   x0, .LC1
20     add    x0, x0, :lo12:.LC1
21     ; найти место в стековом фрейме для переменной "x" (X1=FP+28):
22     add    x1, x29, 28
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

23 ; X1=адрес переменной "x"
24 ; передать адрес в scanf() и вызвать её:
25     bl    __isoc99_scanf
26 ; загрузить 32-битное значение из переменной в стековом фрейме:
27     ldr    w1, [x29,28]
28 ; W1=x
29 ; загрузить указатель на строку "You entered %d...\n"
30 ; printf() возьмет текстовую строку из X0 и переменную "x" из X1 (или W1)
31     adrp   x0, .LC2
32     add    x0, x0, :lo12:.LC2
33     bl    printf
34 ; возврат 0
35     mov    w0, 0
36 ; восстановить FP и LR, затем прибавить 32 к SP:
37     ldp    x29, x30, [sp], 32
38     ret

```

Под стековый фрейм выделяется 32 байта, что больше чем нужно. Может быть, это связано с выравниваем по границе памяти? Самая интересная часть — это поиск места под переменную *x* в стековом фрейме (строка 22). Почему 28? Почему-то, компилятор решил расположить эту переменную в конце стекового фрейма, а не в начале. Адрес потом передается в `scanf()`, которая просто сохраняет значение, введенное пользователем, в памяти по этому адресу. Это 32-битное значение типа *int*. Значение загружается в строке 27 и затем передается в `printf()`.

## MIPS

Для переменной *x* выделено место в стеке, и к нему будут производиться обращения как `$sp + 24`.

Её адрес передается в `scanf()`, а значение прочитанное от пользователя загружается используя инструкцию LW («Load Word» — загрузить слово) и затем оно передается в `printf()`.

Листинг 1.75: Оптимизирующий GCC 4.4.5 (вывод на ассемблере)

```

$LC0:
    .ascii  "Enter X:\000"
$LC1:
    .ascii  "%d\000"
$LC2:
    .ascii  "You entered %d...\012\000"
main:
; пролог функции:
    lui    $28,%hi(__gnu_local_gp)
    addiu  $sp,$sp,-40
    addiu  $28,$28,%lo(__gnu_local_gp)
    sw     $31,36($sp)
; вызов puts():
    lw     $25,%call16(puts)($28)
    lui    $4,%hi($LC0)
    jalr   $25

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        addiu    $4,$4,%lo($LC0) ; branch delay slot
; вызов scanf():
        lw      $28,16($sp)
        lui    $4,%hi($LC1)
        lw      $25,%call16(__isoc99_scanf)($28)
; установить второй аргумент для scanf(), $a1=$sp+24:
        addiu   $5,$sp,24
        jalr   $25
        addiu   $4,$4,%lo($LC1) ; branch delay slot

; вызов printf():
        lw      $28,16($sp)
; установить второй аргумент для printf(),
; загрузить слово по адресу $sp+24:
        lw      $5,24($sp)
        lw      $25,%call16(printf)($28)
        lui    $4,%hi($LC2)
        jalr   $25
        addiu   $4,$4,%lo($LC2) ; branch delay slot

; эпилог функции:
        lw      $31,36($sp)
; установить возвращаемое значение в 0:
        move   $2,$0
; возврат:
        j      $31
        addiu   $sp,$sp,40      ; branch delay slot

```

IDA показывает разметку стека следующим образом:

Листинг 1.76: Оптимизирующий GCC 4.4.5 (IDA)

```

.text:00000000 main:
.text:00000000
.text:00000000 var_18    = -0x18
.text:00000000 var_10    = -0x10
.text:00000000 var_4     = -4
.text:00000000
; пролог функции:
.text:00000000         lui    $gp, (__gnu_local_gp >> 16)
.text:00000004         addiu  $sp, -0x28
.text:00000008         la     $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C         sw    $ra, 0x28+var_4($sp)
.text:00000010         sw    $gp, 0x28+var_18($sp)
; вызов puts():
.text:00000014         lw    $t9, (puts & 0xFFFF)($gp)
.text:00000018         lui  $a0, ($LC0 >> 16) # "Enter X:"
.text:0000001C         jalr $t9
.text:00000020         la   $a0, ($LC0 & 0xFFFF) # "Enter X:" ; branch
        delay slot
; вызов scanf():
.text:00000024         lw    $gp, 0x28+var_18($sp)
.text:00000028         lui  $a0, ($LC1 >> 16) # "%d"
.text:0000002C         lw    $t9, (__isoc99_scanf & 0xFFFF)($gp)

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; установить второй аргумент для scanf(), $a1=$sp+24:
.text:00000030      addiu   $a1, $sp, 0x28+var_10
.text:00000034      jalr   $t9 ; branch delay slot
.text:00000038      la     $a0, ($LC1 & 0xFFFF) # "%d"
; вызов printf():
.text:0000003C      lw     $gp, 0x28+var_18($sp)
; установить второй аргумент для printf(),
; загрузить слово по адресу $sp+24:
.text:00000040      lw     $a1, 0x28+var_10($sp)
.text:00000044      lw     $t9, (printf & 0xFFFF)($gp)
.text:00000048      lui   $a0, ($LC2 >> 16) # "You entered %d...\n"
.text:0000004C      jalr   $t9
.text:00000050      la     $a0, ($LC2 & 0xFFFF) # "You entered %d...\n"
; branch delay slot
; эпилог функции:
.text:00000054      lw     $ra, 0x28+var_4($sp)
; установить возвращаемое значение в 0:
.text:00000058      move  $v0, $zero
; возврат:
.text:0000005C      jr    $ra
.text:00000060      addiu $sp, 0x28 ; branch delay slot

```

### 1.12.2. Классическая ошибка

Это очень популярная ошибка (и/или опечатка) — передать значение `x` вместо указателя на `x`:

```

#include <stdio.h>

int main()
{
    int x;
    printf ("Enter X:\n");

    scanf ("%d", x); // BUG

    printf ("You entered %d...\n", x);

    return 0;
};

```

Что тут происходит? `x` не инициализирована и содержит случайный шум из локального стека. Когда вызывается `scanf()`, ф-ция берет строку от пользователя, парсит её, получает число и пытается записать в `x`, считая его как адрес в памяти. Но там случайный шум, так что `scanf()` будет пытаться писать по случайному адресу. Скорее всего, процесс упадет.

Интересно что некоторые **CRT!**-библиотеки, в отладочной сборке, заполняют только что выделенную память визуальными различимыми числами вроде `0xCCCCCCCC` или `0x0BADF00D`, и т. д. В этом случае, `x` может содержать `0xCCCCCCCC`, и `scanf()` попытается писать по адресу `0xCCCCCCCC`. Если вы заметите что что-то в вашем процессе пытается писать по адресу `0xCCCCCCCC`, вы поймете, что

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

неинициализированная переменная (или указатель) используется без инициализации. Это лучше, чем если только что выделенная память очищается нулевыми байтами.

### 1.12.3. Глобальные переменные

А что если переменная `x` из предыдущего примера будет глобальной переменной, а не локальной? Тогда к ней смогут обращаться из любого другого места, а не только из тела функции. Глобальные переменные считаются [анти-паттерном](#), но ради примера мы можем себе это позволить.

```
#include <stdio.h>

// теперь x это глобальная переменная
int x;

int main()
{
    printf ("Enter X:\n");

    scanf ("%d", &x);

    printf ("You entered %d...\n", x);

    return 0;
};
```

#### MSVC: x86

```
_DATA    SEGMENT
COMM     _x:DWORD
$SG2456  DB    'Enter X:', 0aH, 00H
$SG2457  DB    '%d', 00H
$SG2458  DB    'You entered %d...', 0aH, 00H
_DATA    ENDS
PUBLIC   _main
EXTRN   _scanf:PROC
EXTRN   _printf:PROC
; Function compile flags: /OdtP
_TEXT    SEGMENT
_main    PROC
    push  ebp
    mov   ebp, esp
    push  OFFSET $SG2456
    call  _printf
    add   esp, 4
    push  OFFSET _x
    push  OFFSET $SG2457
    call  _scanf
    add   esp, 8
    mov   eax, DWORD PTR _x
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

push    eax
push    OFFSET $SG2458
call    _printf
add     esp, 8
xor     eax, eax
pop     ebp
ret     0
_main   ENDP
_TEXT   ENDS

```

В целом ничего особенного. Теперь `x` объявлена в сегменте `_DATA`. Память для неё в стеке более не выделяется. Все обращения к ней происходят не через стек, а уже напрямую. Неинициализированные глобальные переменные не занимают места в исполняемом файле (и действительно, зачем в исполняемом файле нужно выделять место под изначально нулевые переменные?), но тогда, когда к этому месту в памяти кто-то обратится, **OS!** подставит туда блок, состоящий из нулей<sup>72</sup>.

Попробуем изменить объявление этой переменной:

```
int x=10; // значение по умолчанию
```

Выйдет в итоге:

```

_DATA   SEGMENT
_x      DD      0aH
...

```

Здесь уже по месту этой переменной записано `0xA` с типом `DD` (dword = 32 бита).

Если вы откроете скомпилированный `.exe`-файл в **IDA!**, то увидите, что `x` находится в начале сегмента `_DATA`, после этой переменной будут текстовые строки.

А вот если вы откроете в **IDA!** `.exe` скомпилированный в прошлом примере, где значение `x` не определено, то вы увидите:

#### Листинг 1.77: IDA!

```

.data:0040FA80 _x          dd ?      ; DATA XREF: _main+10
.data:0040FA80          ; _main+22
.data:0040FA84 dword_40FA84  dd ?      ; DATA XREF: _memset+1E
.data:0040FA84          ; unknown_libname_1+28
.data:0040FA88 dword_40FA88  dd ?      ; DATA XREF: __sbh_find_block+5
.data:0040FA88          ; __sbh_free_block+2BC
.data:0040FA8C ; LPVOID lpMem
.data:0040FA8C lpMem      dd ?      ; DATA XREF: __sbh_find_block+B
.data:0040FA8C          ; __sbh_free_block+2CA
.data:0040FA90 dword_40FA90  dd ?      ; DATA XREF: _V6_HeapAlloc+13
.data:0040FA90          ; __calloc_impl+72
.data:0040FA94 dword_40FA94  dd ?      ; DATA XREF: __sbh_free_block+2FE

```

<sup>72</sup>Так работает **VM!**

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

---

\_x обозначен как ?, наряду с другими переменными не требующими инициализации. Это означает, что при загрузке .exe в память, место под всё это выделено будет и будет заполнено нулевыми байтами [ISO/IEC 9899:TC3 (C C99 standard), (2007)6.7.8p10]. Но в самом .exe ничего этого нет. Неинициализированные переменные не занимают места в исполняемых файлах. Это удобно для больших массивов, например.

## MSVC: x86 + OllyDbg

Тут даже проще:

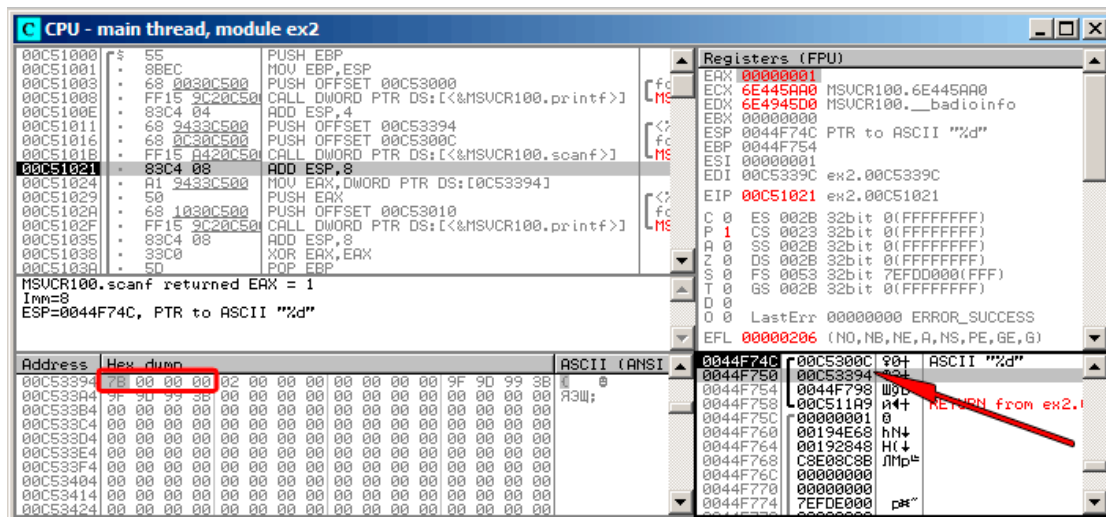


Рис. 1.16: OllyDbg: после исполнения scanf()

Переменная хранится в сегменте данных. Кстати, после исполнения инструкции PUSH (заталкивающей адрес  $x$ ) адрес появится в стеке, и на этом элементе можно нажать правой кнопкой, выбрать «Follow in dump». И в окне памяти слева появится эта переменная.

После того как в консоли введем 123, здесь появится 0x7B.

Почему самый первый байт это 7B? По логике вещей, здесь должно было бы быть 00 00 00 7B. Это называется **endianness**, и в x86 принят формат *little-endian*. Это означает, что в начале записывается самый младший байт, а заканчивается самым старшим байтом. Больше об этом: ?? (стр. ??).

Позже из этого места в памяти 32-битное значение загружается в EAX и передается в printf().

Адрес переменной  $x$  в памяти 0x00C53394.

В OllyDbg мы можем посмотреть карту памяти процесса (Alt-M) и увидим, что этот адрес внутри PE-сегмента `.data` нашей программы:

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00070000	00067000				Map	R	R	C:\Windows\System32\locale.nls
00190000	00005000			Heap	Priv	RW	RW	
00209000	00007000				Priv	RW	Gua	Gua
0044C000	00001000				Priv	RW	RW	Gua
0044D000	00003000			Stack of main thread	Priv	RW	RW	
00590000	00007000				Priv	RW	RW	
00750000	0000C000			Default heap	Priv	RW	RW	
00C50000	00001000	ex2		PE header	Img	R	RWE	Cop
00C51000	00001000	ex2	.text	Code	Img	R E	RWE	Cop
00C52000	00001000	ex2	.rdata	Imports	Img	R	RWE	Cop
00C53000	00001000	ex2	.data	Data	Img	RW	RWE	Cop
00C54000	00001000	ex2	.reloc	Relocations	Img	R	RWE	Cop
6E3E0000	00001000	MSUCR100		PE header	Img	R	RWE	Cop
6E3E1000	00002000	MSUCR100	.text	Code, imports, exports	Img	R E	RWE	Cop
6E493000	00006000	MSUCR100	.data	Data	Img	RW	RWE	Cop
6E499000	00001000	MSUCR100	.rsrc	Resources	Img	R	RWE	Cop
6E49A000	00005000	MSUCR100	.reloc	Relocations	Img	R	RWE	Cop
755D0000	00001000	Mod_755D		PE header	Img	R	RWE	Cop
755D1000	00003000				Img	R E	RWE	Cop
755D4000	00001000				Img	RW	RWE	Cop
755D5000	00003000				Img	R	RWE	Cop
755E0000	00001000	Mod_755E		PE header	Img	R	RWE	Cop
755E1000	00004000				Img	R E	RWE	Cop
7562E000	00005000				Img	RW	RWE	Cop
75633000	00009000				Img	R	RWE	Cop
75640000	00001000	Mod_7564		PE header	Img	R	RWE	Cop
75641000	00003000				Img	R E	RWE	Cop
75679000	00002000				Img	RW	RWE	Cop
7567B000	00004000				Img	R	RWE	Cop
76F50000	00010000	kernel32		PE header	Img	R	RWE	Cop
76F60000	00000000	kernel32	.text	Code, imports, exports	Img	R E	RWE	Cop
77090000	00010000	kernel32	.data	Data	Img	RW	RWE	Cop
770A0000	00010000	kernel32	.rsrc	Resources	Img	R	RWE	Cop
77050000	00000000	kernel32	.reloc	Relocations	Img	R	RWE	Cop
77810000	00001000	KERNELBASE		PE header	Img	R	RWE	Cop
77811000	00004000	KERNELBASE	.text	Code, imports, exports	Img	R E	RWE	Cop
77851000	00002000	KERNELBASE	.data	Data	Img	RW	RWE	Cop
77853000	00001000	KERNELBASE	.rsrc	Resources	Img	R	RWE	Cop
77854000	00003000	KERNELBASE	.reloc	Relocations	Img	R	RWE	Cop
77B20000	00001000	Mod_77B2		PE header	Img	R	RWE	Cop
77B21000	00102000				Img	R E	RWE	Cop
77C23000	00002000				Img	R	RWE	Cop
77C52000	0000C000				Img	RW	RWE	Cop
77C5E000	00006000				Img	R	RWE	Cop
77D00000	00001000	ntdll		PE header	Img	R	RWE	Cop
77D10000	00006000	ntdll	.text	Code, exports	Img	R E	RWE	Cop
77DF0000	00001000	ntdll	RT	Code	Img	R E	RWE	Cop
77E00000	00009000	ntdll	.data	Data	Img	RW	RWE	Cop

Рис. 1.17: OllyDbg: карта памяти процесса

## GCC: x86

В Linux всё почти также. За исключением того, что если значение `x` не определено, то эта переменная будет находиться в сегменте `_bss`. В **ELF!**<sup>73</sup> этот сегмент имеет такие атрибуты:

```
; Segment type: Uninitialized
; Segment permissions: Read/Write
```

Ну а если сделать статическое присвоение этой переменной какого-либо значения, например, 10, то она будет находиться в сегменте `_data`, это сегмент с такими атрибутами:

```
; Segment type: Pure data
; Segment permissions: Read/Write
```

<sup>73</sup>ELF!



**MSVC: x64**

Листинг 1.78: MSVC 2012 x64

```

_DATA SEGMENT
COMM x:DWORD
$SG2924 DB 'Enter X:', 0aH, 00H
$SG2925 DB '%d', 00H
$SG2926 DB 'You entered %d...', 0aH, 00H
_DATA ENDS

_TEXT SEGMENT
main PROC
$LN3:
    sub     rsp, 40

    lea    rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
    call   printf
    lea    rdx, OFFSET FLAT:x
    lea    rcx, OFFSET FLAT:$SG2925 ; '%d'
    call   scanf
    mov    edx, DWORD PTR x
    lea    rcx, OFFSET FLAT:$SG2926 ; 'You entered %d...'
    call   printf

    ; возврат 0
    xor    eax, eax

    add    rsp, 40
    ret    0
main ENDP
_TEXT ENDS

```

Почти такой же код как и в x86. Обратите внимание что для `scanf()` адрес переменной `x` передается при помощи инструкции `LEA`, а во второй `printf()` передается само значение переменной при помощи `MOV`. `DWORD PTR` — это часть языка ассемблера (не имеющая отношения к машинным кодам) показывающая, что тип переменной в памяти именно 32-битный, и инструкция `MOV` должна быть здесь закодирована соответственно.

**ARM: Оптимизирующий Keil 6/2013 (Режим Thumb)**

Листинг 1.79: IDA

```

.text:00000000 ; Segment type: Pure code
.text:00000000 AREA .text, CODE
...
.text:00000000 main
.text:00000000 PUSH {R4,LR}
.text:00000002 ADR R0, aEnterX ; "Enter X:\n"
.text:00000004 BL __2printf
.text:00000008 LDR R1, =x
.text:0000000A ADR R0, aD ; "%d"

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:0000000C      BL      __0scanf
.text:00000010      LDR     R0, =x
.text:00000012      LDR     R1, [R0]
.text:00000014      ADR     R0, aYouEnteredD___ ; "You entered %d...\n"
.text:00000016      BL      __2printf
.text:0000001A      MOVS   R0, #0
.text:0000001C      POP     {R4,PC}
...
.text:00000020 aEnterX DCB "Enter X:",0xA,0 ; DATA XREF: main+2
.text:0000002A      DCB    0
.text:0000002B      DCB    0
.text:0000002C off_2C  DCD x                ; DATA XREF: main+8
.text:0000002C      ; main+10
.text:00000030 aD      DCB "%d",0        ; DATA XREF: main+A
.text:00000033      DCB    0
.text:00000034 aYouEnteredD___ DCB "You entered %d...",0xA,0 ; DATA XREF:
main+14
.text:00000047      DCB    0
.text:00000047 ; .text ends
.text:00000047
...
.data:00000048 ; Segment type: Pure data
.data:00000048      AREA  .data, DATA
.data:00000048      ; ORG 0x48
.data:00000048      EXPORT x
.data:00000048 x      DCD 0xA                ; DATA XREF: main+8
.data:00000048      ; main+10
.data:00000048 ; .data ends

```

Итак, переменная `x` теперь глобальная, и она расположена, почему-то, в другом сегменте, а именно сегменте данных (`.data`). Можно спросить, почему текстовые строки расположены в сегменте кода (`.text`), а `x` нельзя было разместить тут же?

Потому что эта переменная, и как следует из определения, она может меняться. И может быть, меняться часто.

Ну а текстовые строки имеют тип констант, они не будут меняться, поэтому они располагаются в сегменте `.text`.

Сегмент кода иногда может быть расположен в ПЗУ микроконтроллера (не забывайте, мы сейчас имеем дело с встраиваемой (*embedded*) микроэлектроникой, где дефицит памяти — обычное дело), а изменяемые переменные — в ОЗУ.

Хранить в ОЗУ неизменяемые данные, когда в наличии есть ПЗУ, не экономно.

К тому же, сегмент данных в ОЗУ с константами нужно инициализировать перед работой, ведь, после включения ОЗУ, очевидно, она содержит в себе случайную информацию.

Далее мы видим в сегменте кода хранится указатель на переменную `x` (`off_2C`) и все операции с переменной происходят через этот указатель.

Это связано с тем, что переменная `x` может быть расположена где-то довольно

далеко от данного участка кода, так что её адрес нужно сохранить в непосредственной близости к этому коду.

Инструкция LDR в Thumb-режиме может адресовать только переменные в пределах вплоть до 1020 байт от своего местоположения.

Эта же инструкция в ARM-режиме — переменные в пределах  $\pm 4095$  байт.

Таким образом, адрес глобальной переменной *x* нужно расположить в непосредственной близости, ведь нет никакой гарантии, что компоновщик<sup>74</sup> сможет разместить саму переменную где-то рядом, она может быть даже в другом чипе памяти!

Ещё одна вещь: если переменную объявить, как *const*, то компилятор Keil разместит её в сегменте `.constdata`.

Должно быть, впоследствии компоновщик и этот сегмент сможет разместить в ПЗУ вместе с сегментом кода.

## ARM64

Листинг 1.80: Неоптимизирующий GCC 4.9.1 ARM64

```

1  .comm  x,4,4
2  .LC0:
3  .string "Enter X:"
4  .LC1:
5  .string "%d"
6  .LC2:
7  .string "You entered %d...\n"
8  f5:
9  ; сохранить FP и LR в стековом фрейме:
10 stp    x29, x30, [sp, -16]!
11 ; установить стековый фрейм (FP=SP)
12 add    x29, sp, 0
13 ; загрузить указатель на строку "Enter X:":
14 adrp   x0, .LC0
15 add    x0, x0, :lo12:LC0
16 bl     puts
17 ; загрузить указатель на строку "%d":
18 adrp   x0, .LC1
19 add    x0, x0, :lo12:LC1
20 ; сформировать адрес глобальной переменной x:
21 adrp   x1, x
22 add    x1, x1, :lo12:x
23 bl     __isoc99_scanf
24 ; снова сформировать адрес глобальной переменной x:
25 adrp   x0, x
26 add    x0, x0, :lo12:x
27 ; загрузить значение из памяти по этому адресу:
28 ldr    w1, [x0]
29 ; загрузить указатель на строку "You entered %d...\n":
30 adrp   x0, .LC2

```

<sup>74</sup>linker в англоязычной литературе

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

31     add    x0, x0, :lo12:.LC2
32     bl     printf
33 ; возврат 0
34     mov    w0, 0
35 ; восстановить FP и LR:
36     ldp   x29, x30, [sp], 16
37     ret

```

Теперь *x* это глобальная переменная, и её адрес вычисляется при помощи пары инструкций ADRP/ADD (строки 21 и 25).

## MIPS

### Неинициализированная глобальная переменная

Так что теперь переменная *x* глобальная. Сделаем исполняемый файл вместо объектного и загрузим его в **IDA!**. IDA показывает присутствие переменной *x* в ELF-секции `.sbss` (помните о «Global Pointer»? ?? (стр. ??)), так как переменная не инициализируется в самом начале.

Листинг 1.81: Оптимизирующий GCC 4.4.5 (IDA)

```

.text:004006C0 main:
.text:004006C0
.text:004006C0 var_10 = -0x10
.text:004006C0 var_4 = -4
.text:004006C0
; пролог функции:
.text:004006C0     lui     $gp, 0x42
.text:004006C4     addiu  $sp, -0x20
.text:004006C8     li     $gp, 0x418940
.text:004006CC     sw     $ra, 0x20+var_4($sp)
.text:004006D0     sw     $gp, 0x20+var_10($sp)
; вызов puts():
.text:004006D4     la     $t9, puts
.text:004006D8     lui   $a0, 0x40
.text:004006DC     jalr  $t9 ; puts
.text:004006E0     la     $a0, aEnterX      # "Enter X:" ; branch delay
        slot
; вызов scanf():
.text:004006E4     lw     $gp, 0x20+var_10($sp)
.text:004006E8     lui   $a0, 0x40
.text:004006EC     la     $t9, __isoc99_scanf
; подготовить адрес x:
.text:004006F0     la     $a1, x
.text:004006F4     jalr  $t9 ; __isoc99_scanf
.text:004006F8     la     $a0, aD           # "%d" ; branch delay slot
; вызов printf():
.text:004006FC     lw     $gp, 0x20+var_10($sp)
.text:00400700     lui   $a0, 0x40
; взять адрес x:
.text:00400704     la     $v0, x
.text:00400708     la     $t9, printf

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; загрузить значение из переменной "x" и передать его в printf() в $a1:
.text:0040070C      lw      $a1, (x - 0x41099C)($v0)
.text:00400710      jalr   $t9 ; printf
.text:00400714      la     $a0, aYouEnteredD___ # "You entered %d...\n"
; branch delay slot
; эпилог функции:
.text:00400718      lw     $ra, 0x20+var_4($sp)
.text:0040071C      move  $v0, $zero
.text:00400720      jr    $ra
.text:00400724      addiu $sp, 0x20 ; branch delay slot

...

.sbss:0041099C # Segment type: Uninitialized
.sbss:0041099C      .sbss
.sbss:0041099C      .globl x
.sbss:0041099C x:      .space 4
.sbss:0041099C

```

IDA уменьшает количество информации, так что сделаем также листинг используя objdump и добавим туда свои комментарии:

Листинг 1.82: Оптимизирующий GCC 4.4.5 (objdump)

```

1 004006c0 <main>:
2 ; пролог функции:
3 4006c0: 3c1c0042 lui    gp,0x42
4 4006c4: 27bdffe0 addiu  sp,sp,-32
5 4006c8: 279c8940 addiu  gp,gp,-30400
6 4006cc: afbf001c sw    ra,28(sp)
7 4006d0: afbc0010 sw    gp,16(sp)
8 ; вызов puts():
9 4006d4: 8f998034 lw    t9,-32716(gp)
10 4006d8: 3c040040 lui   a0,0x40
11 4006dc: 0320f809 jalr  t9
12 4006e0: 248408f0 addiu  a0,a0,2288 ; branch delay slot
13 ; вызов scanf():
14 4006e4: 8fbc0010 lw    gp,16(sp)
15 4006e8: 3c040040 lui   a0,0x40
16 4006ec: 8f998038 lw    t9,-32712(gp)
17 ; подготовить адрес x:
18 4006f0: 8f858044 lw    a1,-32700(gp)
19 4006f4: 0320f809 jalr  t9
20 4006f8: 248408fc addiu  a0,a0,2300 ; branch delay slot
21 ; вызов printf():
22 4006fc: 8fbc0010 lw    gp,16(sp)
23 400700: 3c040040 lui   a0,0x40
24 ; взять адрес x:
25 400704: 8f828044 lw    v0,-32700(gp)
26 400708: 8f99803c lw    t9,-32708(gp)
27 ; загрузить значение из переменной "x" и передать его в printf() в $a1:
28 40070c: 8c450000 lw    a1,0(v0)
29 400710: 0320f809 jalr  t9
30 400714: 24840900 addiu  a0,a0,2304 ; branch delay slot

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

31 ; эпилог функции:
32 400718: 8fbf001c lw      ra,28(sp)
33 40071c: 00001021 move   v0,zero
34 400720: 03e00008 jr      ra
35 400724: 27bd0020 addiu  sp,sp,32 ; branch delay slot
36 ; набор NOP-ов для выравнивания начала следующей ф-ции по 16-байтной границе:
37 400728: 00200825 move   at,at
38 40072c: 00200825 move   at,at

```

Теперь мы видим, как адрес переменной  $x$  берется из буфера 64KiB, используя GP и прибавление к нему отрицательного смещения (строка 18).

И даже более того: адреса трех внешних функций, используемых в нашем примере (`puts()`, `scanf()`, `printf()`) также берутся из буфера 64KiB используя GP (строки 9, 16 и 26).

GP указывает на середину буфера, так что такие смещения могут нам подсказать, что адреса всех трех функций, а также адрес переменной  $x$  расположены где-то в самом начале буфера. Действительно, ведь наш пример крохотный.

Ещё нужно отметить что функция заканчивается двумя **NOP!**-ами (`MOVE $AT, $AT` — это холостая инструкция), чтобы выровнять начало следующей функции по 16-байтной границе.

### Инициализированная глобальная переменная

Немного изменим наш пример и сделаем, чтобы у  $x$  было значение по умолчанию:

```
int x=10; // значение по умолчанию
```

Теперь IDA показывает что переменная  $x$  располагается в секции `.data`:

Листинг 1.83: Оптимизирующий GCC 4.4.5 (IDA)

```

.text:004006A0 main:
.text:004006A0
.text:004006A0 var_10 = -0x10
.text:004006A0 var_8  = -8
.text:004006A0 var_4  = -4
.text:004006A0
.text:004006A0      lui    $gp, 0x42
.text:004006A4      addiu  $sp, -0x20
.text:004006A8      li     $gp, 0x418930
.text:004006AC      sw    $ra, 0x20+var_4($sp)
.text:004006B0      sw    $s0, 0x20+var_8($sp)
.text:004006B4      sw    $gp, 0x20+var_10($sp)
.text:004006B8      la    $t9, puts
.text:004006BC      lui   $a0, 0x40
.text:004006C0      jalr  $t9 ; puts
.text:004006C4      la    $a0, aEnterX # "Enter X:"
.text:004006C8      lw    $gp, 0x20+var_10($sp)
; подготовить старшую часть адреса x:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:004006CC      lui      $s0, 0x41
.text:004006D0      la       $t9, __isoc99_scanf
.text:004006D4      lui      $a0, 0x40
; прибавить младшую часть адреса x:
.text:004006D8      addiu   $a1, $s0, (x - 0x410000)
; теперь адрес x в $a1.
.text:004006DC      jalr    $t9 ; __isoc99_scanf
.text:004006E0      la      $a0, aD          # "%d"
.text:004006E4      lw      $gp, 0x20+var_10($sp)
; загрузить слово из памяти:
.text:004006E8      lw      $a1, x
; значение x теперь в $a1.
.text:004006EC      la      $t9, printf
.text:004006F0      lui      $a0, 0x40
.text:004006F4      jalr    $t9 ; printf
.text:004006F8      la      $a0, aYouEnteredD___ # "You entered %d...\n"
.text:004006FC      lw      $ra, 0x20+var_4($sp)
.text:00400700      move    $v0, $zero
.text:00400704      lw      $s0, 0x20+var_8($sp)
.text:00400708      jr      $ra
.text:0040070C      addiu   $sp, 0x20

...

.data:00410920      .globl x
.data:00410920 x:      .word 0xA

```

Почему не `.sdata`? Может быть, нужно было указать какую-то опцию в GCC? Тем не менее, `x` теперь в `.data`, а это уже общая память и мы можем посмотреть как происходит работа с переменными там.

Адрес переменной должен быть сформирован парой инструкций. В нашем случае это LUI («Load Upper Immediate» — загрузить старшие 16 бит) и ADDIU («Add Immediate Unsigned Word» — прибавить значение). Вот так же листинг сгенерированный `objdump`-ом для лучшего рассмотрения:

Листинг 1.84: Оптимизирующий GCC 4.4.5 (`objdump`)

```

004006a0 <main>:
 4006a0: 3c1c0042 lui      gp,0x42
 4006a4: 27bdffe0 addiu   sp,sp,-32
 4006a8: 279c8930 addiu   gp,gp,-30416
 4006ac: afbf001c sw      ra,28(sp)
 4006b0: afb00018 sw      s0,24(sp)
 4006b4: afbc0010 sw      gp,16(sp)
 4006b8: 8f998034 lw      t9,-32716(gp)
 4006bc: 3c040040 lui      a0,0x40
 4006c0: 0320f809 jalr    t9
 4006c4: 248408d0 addiu   a0,a0,2256
 4006c8: 8fbc0010 lw      gp,16(sp)
; подготовить старшую часть адреса x:
 4006cc: 3c100041 lui      s0,0x41
 4006d0: 8f998038 lw      t9,-32712(gp)
 4006d4: 3c040040 lui      a0,0x40

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; прибавить младшую часть адреса x:
4006d8: 26050920 addiu  a1,s0,2336
; теперь адрес x в $a1.
4006dc: 0320f809 jalr   t9
4006e0: 248408dc addiu  a0,a0,2268
4006e4: 8fbc0010 lw    gp,16(sp)
; старшая часть адреса x всё ещё в $s0.
; прибавить младшую часть к ней и загрузить слово из памяти:
4006e8: 8e050920 lw    a1,2336(s0)
; значение x теперь в $a1.
4006ec: 8f99803c lw    t9,-32708(gp)
4006f0: 3c040040 lui   a0,0x40
4006f4: 0320f809 jalr   t9
4006f8: 248408e0 addiu  a0,a0,2272
4006fc: 8fbf001c lw    ra,28(sp)
400700: 00001021 move  v0,zero
400704: 8fb00018 lw    s0,24(sp)
400708: 03e00008 jr    ra
40070c: 27bd0020 addiu  sp,sp,32

```

Адрес формируется используя LUI и ADDIU, но старшая часть адреса всё ещё в регистре \$S0, и можно закодировать смещение в инструкции LW («Load Word»), так что одной LW достаточно для загрузки значения из переменной и передачи его в printf(). Регистры хранящие временные данные имеют префикс T-, но здесь есть также регистры с префиксом S-, содержимое которых должно быть сохранено в других функциях (т.е. «saved»).

Вот почему \$S0 был установлен по адресу 0x4006cc и затем был использован по адресу 0x4006e8 после вызова scanf().

Функция scanf() не изменяет это значение.

#### 1.12.4. Проверка результата scanf()

Как уже было упомянуто, использовать scanf() в наше время слегка старомодно. Но если уж пришлось, нужно хотя бы проверять, сработал ли scanf() правильно или пользователь ввел вместо числа что-то другое, что scanf() не смог трактовать как число.

```

#include <stdio.h>

int main()
{
    int x;
    printf ("Enter X:\n");

    if (scanf ("%d", &x)==1)
        printf ("You entered %d...\n", x);
    else
        printf ("What you entered? Huh?\n");

    return 0;
};

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



По стандарту, `scanf()`<sup>75</sup> возвращает количество успешно полученных значений.

В нашем случае, если всё успешно и пользователь ввел таки некое число, `scanf()` вернет 1. А если нет, то 0 (или **EOF!**<sup>76</sup>).

Добавим код, проверяющий результат `scanf()` и в случае ошибки он сообщает пользователю что-то другое.

Это работает предсказуемо:

```
C:\...>ex3.exe
Enter X:
123
You entered 123...

C:\...>ex3.exe
Enter X:
ouch
What you entered? Huh?
```

## MSVC: x86

Вот что выходит на ассемблере (MSVC 2010):

```
    lea    eax, DWORD PTR _x$[ebp]
    push  eax
    push  OFFSET $SG3833 ; '%d', 00H
    call  _scanf
    add   esp, 8
    cmp   eax, 1
    jne   SHORT $LN2@main
    mov   ecx, DWORD PTR _x$[ebp]
    push  ecx
    push  OFFSET $SG3834 ; 'You entered %d...', 0aH, 00H
    call  _printf
    add   esp, 8
    jmp   SHORT $LN1@main
$LN2@main:
    push  OFFSET $SG3836 ; 'What you entered? Huh?', 0aH, 00H
    call  _printf
    add   esp, 4
$LN1@main:
    xor   eax, eax
```

Для того чтобы вызывающая функция имела доступ к результату вызываемой функции, вызываемая функция (в нашем случае `scanf()`) оставляет это значение в регистре EAX.

<sup>75</sup>`scanf`, `wscanf`: [MSDN](#)

<sup>76</sup>**EOF!**

Мы проверяем его инструкцией `CMP EAX, 1` (*CoMPare*), то есть сравниваем значение в `EAX` с `1`.

Следующий за инструкцией `CMP`: условный переход `JNE`. Это означает *Jump if Not Equal*, то есть условный переход *если не равно*.

Итак, если `EAX` не равен `1`, то `JNE` заставит **CPU!** перейти по адресу указанном в операнде `JNE`, у нас это `$LN2@main`. Передав управление по этому адресу, **CPU!** начнет исполнять вызов `printf()` с аргументом `What you entered? Huh?`. Но если всё нормально, перехода не случится и исполнится другой `printf()` с двумя аргументами:

```
'You entered %d...' и значением переменной x.
```

Для того чтобы после этого вызова не исполнился сразу второй вызов `printf()`, после него есть инструкция `JMP`, безусловный переход, который отправит процессор на место после второго `printf()` и перед инструкцией `XOR EAX, EAX`, которая реализует `return 0`.

Итак, можно сказать, что в подавляющих случаях сравнение какой-либо переменной с чем-то другим происходит при помощи пары инструкций `CMP` и `Jcc`, где *cc* это *condition code*. `CMP` сравнивает два значения и выставляет флаги процессора<sup>77</sup>. `Jcc` проверяет нужные ему флаги и выполняет переход по указанному адресу (или не выполняет).

Но на самом деле, как это не парадоксально поначалу звучит, `CMP` это почти то же самое что и инструкция `SUB`, которая отнимает числа одно от другого. Все арифметические инструкции также выставляют флаги в соответствии с результатом, не только `CMP`. Если мы сравним `1` и `1`, от единицы отнимется единица, получится `0`, и выставится флаг `ZF` (*zero flag*), означающий, что последний полученный результат был `0`. Ни при каких других значениях `EAX`, флаг `ZF` не может быть выставлен, кроме тех, когда операнды равны друг другу. Инструкция `JNE` проверяет только флаг `ZF`, и совершает переход только если флаг не поднят. Фактически, `JNE` это синоним инструкции `JNZ` (*Jump if Not Zero*). Ассемблер транслирует обе инструкции в один и тот же опкод. Таким образом, можно `CMP` заменить на `SUB` и всё будет работать также, но разница в том, что `SUB` всё-таки испортит значение в первом операнде. `CMP` это *SUB без сохранения результата, но изменяющая флаги*.

## MSVC: x86: IDA

Наверное, уже пора делать первые попытки анализа кода в **IDA!**. Кстати, начинающим полезно компилировать в `MSVC` с ключом `/MD`, что означает, что все эти стандартные функции не будут скомпонованы с исполняемым файлом, а будут импортироваться из файла `MSVCR*.DLL`. Так будет легче увидеть, где какая стандартная функция используется.

Анализируя код в **IDA!**, очень полезно делать пометки для себя (и других). Например, разбирая этот пример, мы сразу видим, что `JNZ` срабатывает в случае ошибки. Можно навести курсор на эту метку, нажать «n» и переименовать метку в «error». Ещё одну метку — в «exit». Вот как у меня получилось в итоге:

<sup>77</sup>См. также о флагах x86-процессора: [wikipedia](https://en.wikipedia.org/wiki/X86_registers#Flags).

```

.text:00401000 _main proc near
.text:00401000
.text:00401000 var_4 = dword ptr -4
.text:00401000 argc = dword ptr 8
.text:00401000 argv = dword ptr 0Ch
.text:00401000 envp = dword ptr 10h
.text:00401000
.text:00401000     push    ebp
.text:00401001     mov     ebp, esp
.text:00401003     push    ecx
.text:00401004     push    offset Format ; "Enter X:\n"
.text:00401009     call   ds:printf
.text:0040100F     add     esp, 4
.text:00401012     lea    eax, [ebp+var_4]
.text:00401015     push    eax
.text:00401016     push    offset aD ; "%d"
.text:0040101B     call   ds:scanf
.text:00401021     add     esp, 8
.text:00401024     cmp    eax, 1
.text:00401027     jnz    short error
.text:00401029     mov    ecx, [ebp+var_4]
.text:0040102C     push    ecx
.text:0040102D     push    offset aYou ; "You entered %d...\n"
.text:00401032     call   ds:printf
.text:00401038     add     esp, 8
.text:0040103B     jmp    short exit
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D     push    offset aWhat ; "What you entered? Huh?\n"
.text:00401042     call   ds:printf
.text:00401048     add     esp, 4
.text:0040104B exit: ; CODE XREF: _main+3B
.text:0040104B     xor    eax, eax
.text:0040104D     mov    esp, ebp
.text:0040104F     pop    ebp
.text:00401050     retn
.text:00401050 _main endp

```

Так понимать код становится чуть легче. Впрочем, меру нужно знать во всем и комментировать каждую инструкцию не стоит.

В **IDA!** также можно скрывать части функций: нужно выделить скрываемую часть, нажать Ctrl-«-» на цифровой клавиатуре и ввести текст.

Скроем две части и придумаем им названия:

```

.text:00401000 _text segment para public 'CODE' use32
.text:00401000     assume cs:_text
.text:00401000     ;org 401000h
.text:00401000 ; ask for X
.text:00401012 ; get X
.text:00401024     cmp    eax, 1

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
.text:00401027      jnz  short error
.text:00401029      ; print result
.text:0040103B      jmp  short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D      push offset aWhat ; "What you entered? Huh?\n"
.text:00401042      call ds:printf
.text:00401048      add  esp, 4
.text:0040104B
.text:0040104B exit: ; CODE XREF: _main+3B
.text:0040104B      xor  eax, eax
.text:0040104D      mov  esp, ebp
.text:0040104F      pop  ebp
.text:00401050      retn
.text:00401050 _main endp
```

Раскрывать скрытые части функций можно при помощи Ctrl-«+» на цифровой клавиатуре.

Нажав «пробел», мы увидим, как **IDA!** может представить функцию в виде графа:

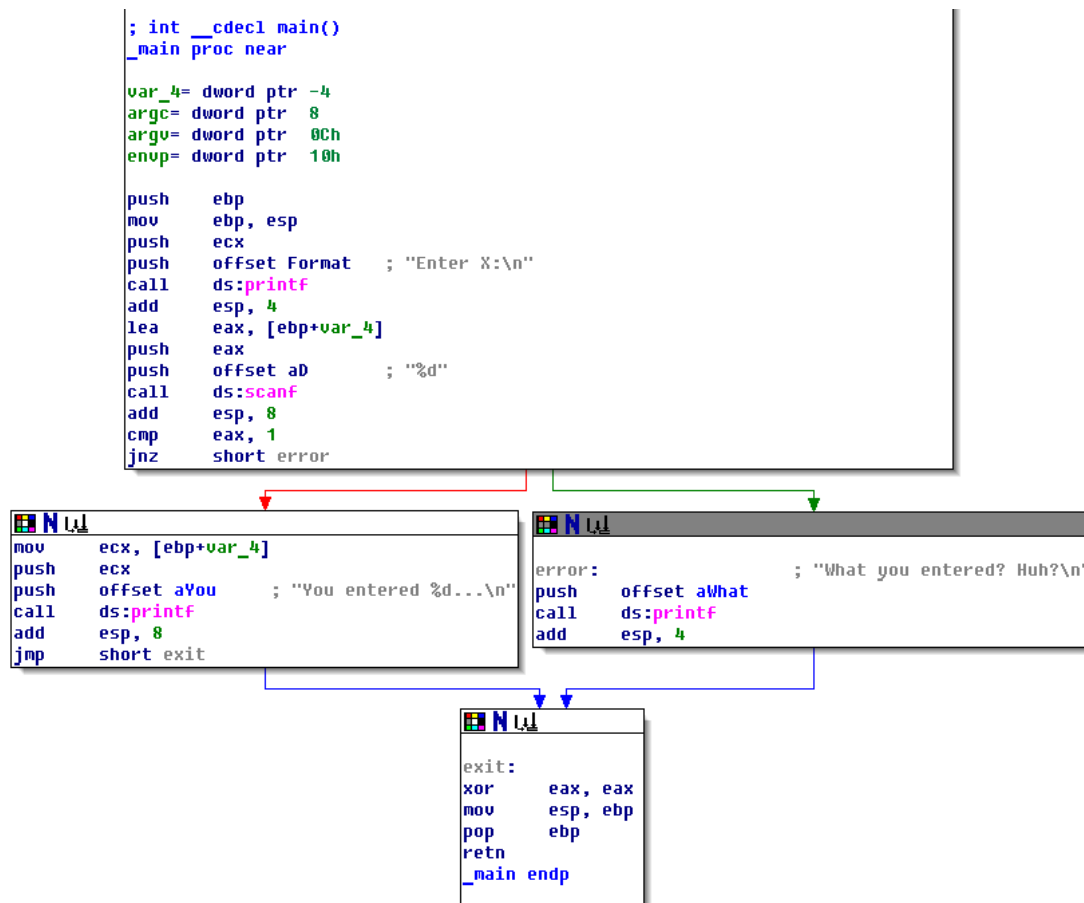


Рис. 1.18: Отображение функции в IDA в виде графа

После каждого условного перехода видны две стрелки: зеленая и красная. Зеленая ведет к тому блоку, который исполнится если переход сработает, а красная — если не сработает.

В этом режиме также можно сворачивать узлы и давать им названия («group nodes»). Сделаем это для трех блоков:

```

; int __cdecl main()
_main proc near

var_4= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
push    ecx
push    offset Format    ; "Enter X:\n"
call   ds:printf
add     esp, 4
lea    eax, [ebp+var_4]
push    eax
push    offset aD        ; "%d"
call   ds:scanf
add     esp, 8
cmp    eax, 1
jnz    short error

```

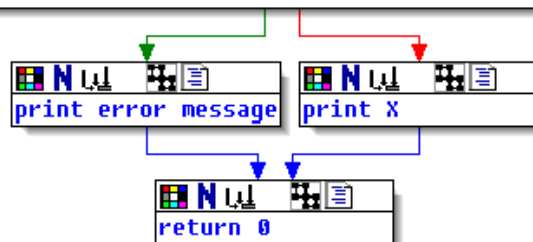


Рис. 1.19: Отображение в IDA в виде графа с тремя свернутыми блоками

Всё это очень полезно делать. Вообще, очень важная часть работы реверсера (да и любого исследователя) состоит в том, чтобы уменьшать количество имеющейся информации.

## MSVC: x86 + OllyDbg

Попробуем в OllyDbg немного хакнуть программу и сделать вид, что `scanf()` срабатывает всегда без ошибок. Когда в `scanf()` передается адрес локальной переменной, изначально в этой переменной находится некий мусор. В данном случае это `0x6E494714`:

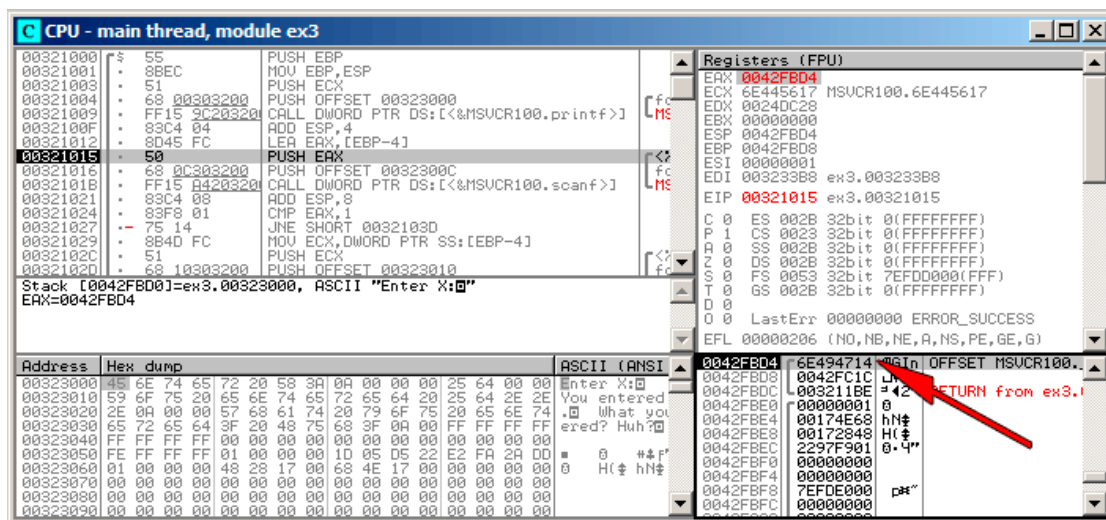


Рис. 1.20: OllyDbg: передача адреса переменной в `scanf()`

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Когда `scanf()` запускается, вводим в консоли что-то непохожее на число, например «`asdasd`». `scanf()` заканчивается с 0 в `EAX`, что означает, что произошла ошибка.

Вместе с этим мы можем посмотреть на локальную переменную в стеке — она не изменилась. Действительно, ведь что туда записала бы функция `scanf()`? Она не делала ничего кроме возвращения нуля. Попробуем ещё немного «хакнуть» нашу программу. Щелкнем правой кнопкой на `EAX`, там, в числе опций, будет также «Set to 1». Это нам и нужно.

В `EAX` теперь 1, последующая проверка пройдет как надо, и `printf()` выведет значение переменной из стека.

Запускаем (F9) и видим в консоли следующее:

Листинг 1.85: консоль

```
Enter X:  
asdasd  
You entered 1850296084...
```

Действительно, 1850296084 это десятичное представление числа в стеке (0x6E494714)!



## MSVC: x86 + Hiew

Это ещё может быть и простым примером исправления исполняемого файла. Мы можем попробовать исправить его таким образом, что программа всегда будет выводить числа, вне зависимости от ввода.

Исполняемый файл скомпилирован с импортированием функций из MSVCR\*.DLL (т.е. с опцией /MD)<sup>78</sup>, поэтому мы можем отыскать функцию main() в самом начале секции .text. Откроем исполняемый файл в Hiew, найдем самое начало секции .text (Enter, F8, F6, Enter, Enter).

Мы увидим следующее:

```

Hiew: ex3.exe
C:\Polygon\ollydbg\ex3.exe  FRO ----- a32 PE .00401000 Hie
.00401000: 55          push     ebp
.00401001: 8BEC       mov     ebp,esp
.00401003: 51          push     ecx
.00401004: 6800304000 push     000403000 ;'Enter X:' --[1]
.00401009: FF1594204000 call    printf
.0040100F: 83C404     add     esp,4
.00401012: 8D45FC     lea    eax,[ebp][-4]
.00401015: 50          push     eax
.00401016: 680C304000 push     00040300C --[2]
.0040101B: FF158C204000 call    scanf
.00401021: 83C408     add     esp,8
.00401024: 83F801     cmp     eax,1
.00401027: 7514       jnz     .00040103D --[3]
.00401029: 8B4DFC     mov     ecx,[ebp][-4]
.0040102C: 51          push     ecx
.0040102D: 6810304000 push     000403010 ;'You entered %d...'
.00401032: FF1594204000 call    printf
.00401038: 83C408     add     esp,8
.0040103B: EB0E       jmps    .00040104B --[5]
.0040103D: 6824304000 3push   000403024 ;'What you entered?'
.00401042: FF1594204000 call    printf
.00401048: 83C404     add     esp,4
.0040104B: 33C0       5xor    eax,eax
.0040104D: 8BE5       mov     esp,ebp
.0040104F: 5D          pop     ebp
.00401050: C3         retn   ; ^^^^
.00401051: B84D5A0000 mov     eax,00005A4D ;' ZM'
1Global 2FileBlk 3CryBlk 4Reload 5OrdLdr 6String 7Direct 8Table 9Ibyte 10Leave 11Nak

```

Рис. 1.21: Hiew: функция main()

Hiew находит **ASCIIZ!**<sup>79</sup>-строки и показывает их, также как и имена импортируемых функций.

<sup>78</sup>То, что ещё называют «dynamic linking»

<sup>79</sup>**ASCIIZ!**

Переведите курсор на адрес .00401027 (с инструкцией JNZ, которую мы хотим заблокировать), нажмите F3, затем наберите «9090» (что означает два **NOP!**-а):

```

Hiew: ex3.exe
C:\Polygon\ollydbg\ex3.exe  FFWO EDITMODE  a32 PE  0000
00000400: 55      push   ebp
00000401: 8BEC   mov    ebp,esp
00000403: 51     push   ecx
00000404: 6800304000  push  000403000 ;' @0 '
00000409: FF1594204000  call  d,[000402094]
0000040F: 83C404  add    esp,4
00000412: 8D45FC  lea   eax,[ebp][-4]
00000415: 50     push   eax
00000416: 680C304000  push  00040300C ;' @00 '
0000041B: FF158C204000  call  d,[00040208C]
00000421: 83C408  add    esp,8
00000424: 83F801  cmp    eax,1
00000427: 90     nop
00000428: 90     nop
00000429: 8B4DFC  mov    ecx,[ebp][-4]
0000042C: 51     push   ecx
0000042D: 6810304000  push  000403010 ;' @00 '
00000432: FF1594204000  call  d,[000402094]
00000438: 83C408  add    esp,8
0000043B: EB0E   jmps  00000044B
0000043D: 6824304000  push  000403024 ;' @0$ '
00000442: FF1594204000  call  d,[000402094]
00000448: 83C404  add    esp,4
0000044B: 33C0   xor    eax,eax
0000044D: 8BE5   mov    esp,ebp
0000044F: 5D     pop    ebp
00000450: C3     retn  ; _^_ ^_ ^_ ^_ ^_ ^_ ^_ ^_ ^_ ^_
1      2Nops  3      4      5      6      7      8Table 9      10

```

Рис. 1.22: Hiew: замена JNZ на два **NOP!**-а

Затем F9 (update). Теперь исполняемый файл записан на диск. Он будет вести себя так, как нам надо.

Два **NOP!**-а, возможно, не так эстетично, как могло бы быть. Другой способ изменить инструкцию это записать 0 во второй байт опкода (смещение перехода), так что JNZ всегда будет переходить на следующую инструкцию.

Можно изменить и наоборот: первый байт заменить на EB, второй байт (смещение перехода) не трогать. Получится всегда срабатывающий безусловный переход. Теперь сообщение об ошибке будет выдаваться всегда, даже если мы ввели число.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

**MSVC: x64**

Так как здесь мы работаем с переменными типа *int*, а они в x86-64 остались 32-битными, то мы здесь видим, как продолжают использоваться регистры с префиксом E-. Но для работы с указателями, конечно, используются 64-битные части регистров с префиксом R-.

Листинг 1.86: MSVC 2012 x64

```

_DATA    SEGMENT
$SG2924 DB      'Enter X:', 0aH, 00H
$SG2926 DB      '%d', 00H
$SG2927 DB      'You entered %d...', 0aH, 00H
$SG2929 DB      'What you entered? Huh?', 0aH, 00H
_DATA    ENDS

_TEXT    SEGMENT
x$ = 32
main     PROC
$LN5:
        sub     rsp, 56
        lea    rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
        call   printf
        lea    rdx, QWORD PTR x$[rsp]
        lea    rcx, OFFSET FLAT:$SG2926 ; '%d'
        call   scanf
        cmp    eax, 1
        jne    SHORT $LN2@main
        mov    edx, DWORD PTR x$[rsp]
        lea    rcx, OFFSET FLAT:$SG2927 ; 'You entered %d...'
        call   printf
        jmp    SHORT $LN1@main
$LN2@main:
        lea    rcx, OFFSET FLAT:$SG2929 ; 'What you entered? Huh?'
        call   printf
$LN1@main:
        ; возврат 0
        xor    eax, eax
        add    rsp, 56
        ret    0
main     ENDP
_TEXT    ENDS
END

```

**ARM****ARM: Оптимизирующий Keil 6/2013 (Режим Thumb)**

Листинг 1.87: Оптимизирующий Keil 6/2013 (Режим Thumb)

```

var_8    = -8

        PUSH    {R3,LR}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

ADR    R0, aEnterX      ; "Enter X:\n"
BL     __2printf
MOV    R1, SP
ADR    R0, aD           ; "%d"
BL     __0scanf
CMP    R0, #1
BEQ    loc_1E
ADR    R0, aWhatYouEntered ; "What you entered? Huh?\n"
BL     __2printf

loc_1A                                ; CODE XREF: main+26
MOV    R0, #0
POP    {R3,PC}

loc_1E                                ; CODE XREF: main+12
LDR    R1, [SP,#8+var_8]
ADR    R0, aYouEnteredD___ ; "You entered %d...\n"
BL     __2printf
B      loc_1A

```

Здесь для нас есть новые инструкции: **CMR** и **BEQ!**<sup>80</sup>.

**CMR** аналогична той что в x86: она отнимает один аргумент от второго и сохраняет флаги.

**BEQ!** совершает переход по другому адресу, если операнды при сравнении были равны, либо если результат последнего вычисления был 0, либо если флаг Z равен 1. То же что и JZ в x86.

Всё остальное просто: исполнение разветвляется на две ветки, затем они сходятся там, где в R0 записывается 0 как возвращаемое из функции значение и происходит выход из функции.

## ARM64

Листинг 1.88: Неоптимизирующий GCC 4.9.1 ARM64

```

1  .LC0:
2      .string "Enter X:"
3  .LC1:
4      .string "%d"
5  .LC2:
6      .string "You entered %d...\n"
7  .LC3:
8      .string "What you entered? Huh?"
9  f6:
10 ; сохранить FP и LR в стековом фрейме:
11     stp    x29, x30, [sp, -32]!
12 ; установить стековый фрейм (FP=SP)
13     add    x29, sp, 0
14 ; загрузить указатель на строку "Enter X:"

```

<sup>80</sup>**BEQ!**

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

15     adrp    x0, .LC0
16     add     x0, x0, :lo12:LC0
17     bl     puts
18 ; загрузить указатель на строку "%d":
19     adrp    x0, .LC1
20     add     x0, x0, :lo12:LC1
21 ; вычислить адрес переменной x в локальном стеке
22     add     x1, x29, 28
23     bl     __isoc99_scanf
24 ; scanf() возвращает результат в W0.
25 ; проверяем его:
26     cmp     w0, 1
27 ; BNE это Branch if Not Equal (переход, если не равно)
28 ; так что если W0<>1, произойдет переход на L2
29     bne     .L2
30 ; в этот момент W0=1, означая, что ошибки не было
31 ; загрузить значение x из локального стека
32     ldr     w1, [x29,28]
33 ; загрузить указатель на строку "You entered %d...\n":
34     adrp    x0, .LC2
35     add     x0, x0, :lo12:LC2
36     bl     printf
37 ; пропустить код, печатающий строку "What you entered? Huh?":
38     b     .L3
39 .L2:
40 ; загрузить указатель на строку "What you entered? Huh?":
41     adrp    x0, .LC3
42     add     x0, x0, :lo12:LC3
43     bl     puts
44 .L3:
45 ; возврат 0
46     mov     w0, 0
47 ; восстановить FP и LR:
48     ldp     x29, x30, [sp], 32
49     ret

```

Исполнение здесь разветвляется, используя пару инструкций CMP/BNE (Branch if Not Equal: переход если не равно).

## MIPS

Листинг 1.89: Оптимизирующий GCC 4.4.5 (IDA)

```

.text:004006A0 main:
.text:004006A0
.text:004006A0 var_18    = -0x18
.text:004006A0 var_10    = -0x10
.text:004006A0 var_4     = -4
.text:004006A0
.text:004006A0         lui     $gp, 0x42
.text:004006A4         addiu  $sp, -0x28
.text:004006A8         li     $gp, 0x418960
.text:004006AC         sw     $ra, 0x28+var_4($sp)

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:004006B0      sw      $gp, 0x28+var_18($sp)
.text:004006B4      la      $t9, puts
.text:004006B8      lui     $a0, 0x40
.text:004006BC      jalr   $t9 ; puts
.text:004006C0      la      $a0, aEnterX      # "Enter X:"
.text:004006C4      lw      $gp, 0x28+var_18($sp)
.text:004006C8      lui     $a0, 0x40
.text:004006CC      la      $t9, __isoc99_scanf
.text:004006D0      la      $a0, aD           # "%d"
.text:004006D4      jalr   $t9 ; __isoc99_scanf
.text:004006D8      addiu  $a1, $sp, 0x28+var_10 # branch delay slot
.text:004006DC      li      $v1, 1
.text:004006E0      lw      $gp, 0x28+var_18($sp)
.text:004006E4      beq    $v0, $v1, loc_40070C
.text:004006E8      or     $at, $zero        # branch delay slot, NOP
.text:004006EC      la      $t9, puts
.text:004006F0      lui     $a0, 0x40
.text:004006F4      jalr   $t9 ; puts
.text:004006F8      la      $a0, aWhatYouEntered # "What you entered?
      Huh?"
.text:004006FC      lw      $ra, 0x28+var_4($sp)
.text:00400700      move   $v0, $zero
.text:00400704      jr     $ra
.text:00400708      addiu  $sp, 0x28

.text:0040070C loc_40070C:
.text:0040070C      la      $t9, printf
.text:00400710      lw      $a1, 0x28+var_10($sp)
.text:00400714      lui     $a0, 0x40
.text:00400718      jalr   $t9 ; printf
.text:0040071C      la      $a0, aYouEnteredD___ # "You entered
      %d...\n"
.text:00400720      lw      $ra, 0x28+var_4($sp)
.text:00400724      move   $v0, $zero
.text:00400728      jr     $ra
.text:0040072C      addiu  $sp, 0x28

```

scanf() возвращает результат своей работы в регистре \$V0 и он проверяется по адресу 0x004006E4 сравнивая значения в \$V0 и \$V1 (1 записан в \$V1 ранее, на 0x004006DC). BEQ означает «Branch Equal» (переход если равно). Если значения равны (т.е. в случае успеха), произойдет переход по адресу 0x0040070C.

### Упражнение

Как мы можем увидеть, инструкцию JNE/JNZ можно вполне заменить на JE/JZ или наоборот (или BNE на BEQ и наоборот). Но при этом ещё нужно переставить базовые блоки местами. Попробуйте сделать это в каком-нибудь примере.

#### 1.12.5. Упражнение

- <http://challenges.re/53>

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

## 1.13. Стоит отметить: глобальные и локальные переменные

Теперь вы знаете, что глобальные переменные обнуляются в начале в **OS!** (?? (стр. ??), [ISO/IEC 9899:TC3 (C C99 standard), (2007)6.7.8p10]), а локальные – нет (?? (стр. ??)).

Иногда, у вас есть глобальная переменная, которую вы забыли проинициализировать, и исполнение вашей программы зависит от того факта, что в начале исполнения там ноль. Потом вы редактируете вашу программу и перемещаете глобальную переменную внутрь ф-ции, делая её локальной. Она не будет более инициализироваться в ноль, и это может в итоге приводить к труднонаходимым ошибкам.

## 1.14. Доступ к переданным аргументам

Как мы уже успели заметить, вызывающая функция передает аргументы для вызываемой через стек. А как вызываемая функция получает к ним доступ?

Листинг 1.90: простой пример

```
#include <stdio.h>

int f (int a, int b, int c)
{
    return a*b+c;
};

int main()
{
    printf ("%d\n", f(1, 2, 3));
    return 0;
};
```

### 1.14.1. x86

#### MSVC

Рассмотрим пример, скомпилированный в (MSVC 2010 Express):

Листинг 1.91: MSVC 2010 Express

```
_TEXT    SEGMENT
_a$ = 8      ; size = 4
_b$ = 12     ; size = 4
_c$ = 16     ; size = 4
_f        PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    imul   eax, DWORD PTR _b$[ebp]
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        add     eax, DWORD PTR _c$[ebp]
        pop     ebp
        ret     0
_f      ENDP

_main  PROC
        push   ebp
        mov    ebp, esp
        push   3 ; третий аргумент
        push   2 ; второй аргумент
        push   1 ; первый аргумент
        call  _f
        add    esp, 12
        push   eax
        push   OFFSET $SG2463 ; '%d', 0aH, 00H
        call  _printf
        add    esp, 8
        ; возврат 0
        xor    eax, eax
        pop    ebp
        ret    0
_main  ENDP

```

Итак, здесь видно: в функции `main()` заталкиваются три числа в стек и вызывается функция `f(int,int,int)`.

Внутри `f()` доступ к аргументам, также как и к локальным переменным, происходит через макросы: `_a$ = 8`, но разница в том, что эти смещения со знаком *плюс*, таким образом если прибавить макрос `_a$` к указателю на `EBP`, то адресуется *внешняя* часть **фрейма** стека относительно `EBP`.

Далее всё более-менее просто: значение `a` помещается в `EAX`. Далее `EAX` умножается при помощи инструкции `IMUL` на то, что лежит в `_b`, и в `EAX` остается **произведение** этих двух значений.

Далее к регистру `EAX` прибавляется то, что лежит в `_c`.

Значение из `EAX` никуда не нужно перекладывать, оно уже лежит где надо. Возвращаем управление вызывающей функции — она возьмет значение из `EAX` и отправит его в `printf()`.

## MSVC + OllyDbg

Проиллюстрируем всё это в `OllyDbg`. Когда мы протрассируем до первой инструкции в `f()`, которая использует какой-то из аргументов (первый), мы увидим, что `EBP` указывает на **фрейм стека**. Он выделен красным прямоугольником.

Самый первый элемент **фрейма стека** — это сохраненное значение `EBP`, затем **RA!**. Третий элемент это первый аргумент функции, затем второй аргумент и третий.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



Для доступа к первому аргументу функции нужно прибавить к EBP 8 (2 32-битных слова).

OllYDbg в курсе этого, так что он добавил комментарии к элементам стека вроде «RETURN from» и «Arg1 = ...», и т. д.

N.B.: аргументы функции являются членами фрейма стека вызывающей функции, а не текущей. Поэтому OllYDbg отметил элементы «Arg» как члены другого фрейма стека.

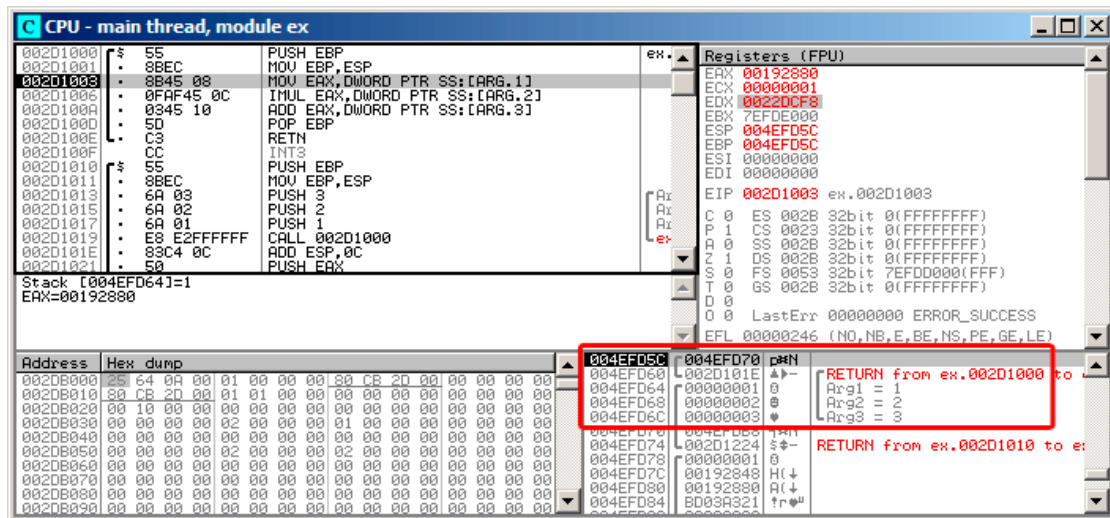


Рис. 1.23: OllYDbg: внутри функции f()

## GCC

Скомпилируем то же в GCC 4.4.1 и посмотрим результат в IDA!:

Листинг 1.92: GCC 4.4.1

```

public f
f
proc near

arg_0 = dword ptr 8
arg_4 = dword ptr 0Ch
arg_8 = dword ptr 10h

push    ebp
mov     ebp, esp
mov     eax, [ebp+arg_0] ; первый аргумент
imul   eax, [ebp+arg_4] ; второй аргумент
add    eax, [ebp+arg_8] ; третий аргумент
pop     ebp
retn
f
endp

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

main      public main
          proc near

var_10    = dword ptr -10h
var_C     = dword ptr -0Ch
var_8     = dword ptr -8

          push    ebp
          mov     ebp, esp
          and     esp, 0FFFFFFF0h
          sub     esp, 10h
          mov     [esp+10h+var_8], 3 ; третий аргумент
          mov     [esp+10h+var_C], 2 ; второй аргумент
          mov     [esp+10h+var_10], 1 ; первый аргумент
          call    f
          mov     edx, offset aD ; "%d\n"
          mov     [esp+10h+var_C], eax
          mov     [esp+10h+var_10], edx
          call    _printf
          mov     eax, 0
          leave
          retn
main      endp

```

Практически то же самое, если не считать мелких отличий описанных ранее.

После вызова обеих функций [указатель стека](#) не возвращается назад, потому что предпоследняя инструкция LEAVE (?? (стр. ??)) делает это за один раз, в конце исполнения.

### 1.14.2. x64

В x86-64 всё немного иначе, здесь аргументы функции (4 или 6) передаются через регистры, а [callee](#) читает их из регистров, а не из стека.

### MSVC

Оптимизирующий MSVC:

Листинг 1.93: Оптимизирующий MSVC 2012 x64

```

$SG2997 DB      '%d', 0aH, 00H

main          PROC
              sub     rsp, 40
              mov     edx, 2
              lea    r8d, QWORD PTR [rdx+1] ; R8D=3
              lea    ecx, QWORD PTR [rdx-1] ; ECX=1
              call   f
              lea    rcx, OFFSET FLAT:$SG2997 ; '%d'
              mov     edx, eax
              call   printf
              xor     eax, eax

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    add    rsp, 40
    ret    0
main    ENDP

f      PROC
    ; ECX - первый аргумент
    ; EDX - второй аргумент
    ; R8D - третий аргумент
    imul  ecx, edx
    lea   eax, DWORD PTR [r8+rcx]
    ret   0
f      ENDP

```

Как видно, очень компактная функция `f()` берет аргументы прямо из регистров.

Инструкция `LEA` используется здесь для сложения чисел. Должно быть компилятор посчитал, что это будет эффективнее использования `ADD`.

В самой `main()` `LEA` также используется для подготовки первого и третьего аргумента: должно быть, компилятор решил, что `LEA` будет работать здесь быстрее, чем загрузка значения в регистр при помощи `MOV`.

Попробуем посмотреть вывод неоптимизирующего `MSVC`:

Листинг 1.94: `MSVC 2012 x64`

```

f      proc near
; область "shadow":
arg_0   = dword ptr  8
arg_8   = dword ptr 10h
arg_10  = dword ptr 18h

    ; ECX - первый аргумент
    ; EDX - второй аргумент
    ; R8D - третий аргумент
    mov     [rsp+arg_10], r8d
    mov     [rsp+arg_8], edx
    mov     [rsp+arg_0], ecx
    mov     eax, [rsp+arg_0]
    imul   eax, [rsp+arg_8]
    add    eax, [rsp+arg_10]
    retn
f      endp

main   proc near
    sub    rsp, 28h
    mov    r8d, 3 ; третий аргумент
    mov    edx, 2 ; второй аргумент
    mov    ecx, 1 ; первый аргумент
    call   f
    mov    edx, eax
    lea   rcx, $SG2931 ; "%d\n"
    call  printf

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; возврат 0
xor     eax, eax
add     rsp, 28h
retn
main   endp

```

Немного путаннее: все 3 аргумента из регистров зачем-то сохраняются в стеке. Это называется «shadow space»<sup>81</sup>: каждая функция в Win64 может (хотя и не обязана) сохранять значения 4-х регистров там.

Это делается по крайней мере из-за двух причин: 1) в большой функции отвести целый регистр (а тем более 4 регистра) для входного аргумента слишком расточительно, так что к нему будет обращение через стек;

2) отладчик всегда знает, где найти аргументы функции в момент останова<sup>82</sup>.

Так что, какие-то большие функции могут сохранять входные аргументы в «shadow space» для использования в будущем, а небольшие функции, как наша, могут этого и не делать.

Место в стеке для «shadow space» выделяет именно [caller](#).

## GCC

Оптимизирующий GCC также делает понятный код:

Листинг 1.95: Оптимизирующий GCC 4.4.6 x64

```

f:
; EDI - первый аргумент
; ESI - второй аргумент
; EDX - третий аргумент
imul   esi, edi
lea    eax, [rdx+rsi]
ret

main:
sub    rsp, 8
mov    edx, 3
mov    esi, 2
mov    edi, 1
call   f
mov    edi, OFFSET FLAT:.LC0 ; "%d\n"
mov    esi, eax
xor    eax, eax ; количество переданных векторных регистров
call   printf
xor    eax, eax
add    rsp, 8
ret

```

<sup>81</sup>[MSDN](#)

<sup>82</sup>[MSDN](#)

Неоптимизирующий GCC:

Листинг 1.96: GCC 4.4.6 x64

```
f:
; EDI - первый аргумент
; ESI - второй аргумент
; EDX - третий аргумент
push    rbp
mov     rbp, rsp
mov     DWORD PTR [rbp-4], edi
mov     DWORD PTR [rbp-8], esi
mov     DWORD PTR [rbp-12], edx
mov     eax, DWORD PTR [rbp-4]
imul   eax, DWORD PTR [rbp-8]
add    eax, DWORD PTR [rbp-12]
leave
ret

main:
push    rbp
mov     rbp, rsp
mov     edx, 3
mov     esi, 2
mov     edi, 1
call   f
mov     edx, eax
mov     eax, OFFSET FLAT:.LC0 ; "%d\n"
mov     esi, edx
mov     rdi, rax
mov     eax, 0 ; количество переданных векторных регистров
call   printf
mov     eax, 0
leave
ret
```

В соглашении о вызовах System V \*NIX ([Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)]<sup>83</sup>) нет «shadow space», но *callee* тоже иногда должен сохранять где-то аргументы, потому что, опять же, регистров может и не хватить на все действия. Что мы здесь и видим.

### **GCC: uint64\_t вместо int**

Наш пример работал с 32-битным *int*, поэтому использовались 32-битные части регистров с префиксом E-.

Его можно немного переделать, чтобы он заработал с 64-битными значениями:

```
#include <stdio.h>
#include <stdint.h>
```

<sup>83</sup>Также доступно здесь: <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
    return a*b+c;
};

int main()
{
    printf ("%lld\n", f(0x1122334455667788,
                        0x1111111122222222,
                        0x3333333344444444));
    return 0;
};

```

Листинг 1.97: Оптимизирующий GCC 4.4.6 x64

```

f      proc near
      imul   rsi, rdi
      lea   rax, [rdx+rsi]
      retn
f      endp

main   proc near
      sub    rsp, 8
      mov   rdx, 3333333344444444h ; третий аргумент
      mov   rsi, 1111111122222222h ; второй аргумент
      mov   rdi, 1122334455667788h ; первый аргумент
      call  f
      mov   edi, offset format ; "%lld\n"
      mov   rsi, rax
      xor   eax, eax ; количество переданных векторных регистров
      call  _printf
      xor   eax, eax
      add   rsp, 8
      retn
main   endp

```

Собственно, всё то же самое, только используются регистры *целиком*, с префиксом R-.

### 1.14.3. ARM

#### Неоптимизирующий Keil 6/2013 (Режим ARM)

```

.text:000000A4 00 30 A0 E1      MOV     R3, R0
.text:000000A8 93 21 20 E0      MLA    R0, R3, R1, R2
.text:000000AC 1E FF 2F E1      BX     LR
...
.text:000000B0                main
.text:000000B0 10 40 2D E9      STMFD  SP!, {R4,LR}
.text:000000B4 03 20 A0 E3      MOV    R2, #3
.text:000000B8 02 10 A0 E3      MOV    R1, #2
.text:000000BC 01 00 A0 E3      MOV    R0, #1

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
.text:000000C0 F7 FF FF EB    BL    f
.text:000000C4 00 40 A0 E1    MOV   R4, R0
.text:000000C8 04 10 A0 E1    MOV   R1, R4
.text:000000CC 5A 0F 8F E2    ADR   R0, aD_0      ; "%d\n"
.text:000000D0 E3 18 00 EB    BL    __2printf
.text:000000D4 00 00 A0 E3    MOV   R0, #0
.text:000000D8 10 80 BD E8    LDMFD SP!, {R4,PC}
```

В функции `main()` просто вызываются две функции, в первую (`f()`) передается три значения. Как уже было упомянуто, первые 4 значения в ARM обычно передаются в первых 4-х регистрах (R0-R3). Функция `f()`, как видно, использует три первых регистра (R0-R2) как аргументы.

Инструкция `MLA` (*Multiply Accumulate*) перемножает два первых операнда (R3 и R1), прибавляет к произведению третий операнд (R2) и помещает результат в нулевой регистр (R0), через который, по стандарту, возвращаются значения функций.

Умножение и сложение одновременно (*Fused multiply-add*) это часто применяемая операция. Кстати, аналогичной инструкции в x86 не было до появления FMA-инструкций в SIMD <sup>84</sup>.

Самая первая инструкция `MOV R3, R0`, по-видимому, избыточна (можно было бы обойтись только одной инструкцией `MLA`). Компилятор не оптимизировал её, ведь, это компиляция без оптимизации.

Инструкция `BX` возвращает управление по адресу, записанному в **LR!** и, если нужно, переключает режимы процессора с Thumb на ARM или наоборот. Это может быть необходимым потому, что, как мы видим, функции `f()` неизвестно, из какого кода она будет вызываться, из ARM или Thumb. Поэтому, если она будет вызываться из кода Thumb, `BX` не только возвращает управление в вызывающую функцию, но также переключает процессор в режим Thumb. Либо не переключит, если функция вызывалась из кода для режима ARM: [ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition, (2012)A2.3.2].

### Оптимизирующий Keil 6/2013 (Режим ARM)

```
.text:00000098                f
.text:00000098 91 20 20 E0    MLA   R0, R1, R0, R2
.text:0000009C 1E FF 2F E1    BX    LR
```

А вот и функция `f()`, скомпилированная компилятором Keil в режиме полной оптимизации (-O3). Инструкция `MOV` была оптимизирована: теперь `MLA` использует все входящие регистры и помещает результат в R0, где вызываемая функция будет его читать и использовать.

### Оптимизирующий Keil 6/2013 (Режим Thumb)

<sup>84</sup>[wikipedia](#)

.text:0000005E	48 43	MULS	R0, R1
.text:00000060	80 18	ADDS	R0, R0, R2
.text:00000062	70 47	BX	LR

В режиме Thumb инструкция MLA недоступна, так что компилятору пришлось сгенерировать код, делающий обе операции по отдельности.

Первая инструкция MULS умножает R0 на R1, оставляя результат в R0. Вторая (ADDS) складывает результат и R2, оставляя результат в R0.

## ARM64

### Оптимизирующий GCC (Linaro) 4.9

Тут всё просто. MADD это просто инструкция, производящая умножение и сложение одновременно (как MLA, которую мы уже видели). Все 3 аргумента передаются в 32-битных частях X-регистров. Действительно, типы аргументов это 32-битные *int*'ы. Результат возвращается в W0.

Листинг 1.98: Оптимизирующий GCC (Linaro) 4.9

```
f:
    madd    w0, w0, w1, w2
    ret

main:
; сохранить FP и LR в стековом фрейме:
    stp    x29, x30, [sp, -16]!
    mov    w2, 3
    mov    w1, 2
    add    x29, sp, 0
    mov    w0, 1
    bl    f
    mov    w1, w0
    adrp   x0, .LC7
    add    x0, x0, :lo12:LC7
    bl    printf
; возврат 0
    mov    w0, 0
; восстановить FP и LR
    ldp    x29, x30, [sp], 16
    ret

.LC7:
    .string "%d\n"
```

Также расширим все типы данных до 64-битных `uint64_t` и попробуем:

```
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

{
    return a*b+c;
};

int main()
{
    printf ("%lld\n", f(0x1122334455667788,
                        0x1111111122222222,
                        0x3333333344444444));
    return 0;
};

```

```

f:
    madd    x0, x0, x1, x2
    ret
main:
    mov     x1, 13396
    adrp   x0, .LC8
    stp    x29, x30, [sp, -16]!
    movk   x1, 0x27d0, lsl 16
    add    x0, x0, :lo12:.LC8
    movk   x1, 0x122, lsl 32
    add    x29, sp, 0
    movk   x1, 0x58be, lsl 48
    bl     printf
    mov    w0, 0
    ldp    x29, x30, [sp], 16
    ret

.LC8:
    .string "%lld\n"

```

Функция `f()` точно такая же, только теперь используются полные части 64-битных X-регистров. Длинные 64-битные значения загружаются в регистры по частям, это описано здесь: ?? (стр. ??).

### Неоптимизирующий GCC (Linaro) 4.9

Неоптимизирующий компилятор выдает немного лишнего кода:

```

f:
    sub     sp, sp, #16
    str     w0, [sp,12]
    str     w1, [sp,8]
    str     w2, [sp,4]
    ldr     w1, [sp,12]
    ldr     w0, [sp,8]
    mul     w1, w1, w0
    ldr     w0, [sp,4]
    add     w0, w1, w0
    add     sp, sp, 16
    ret

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Код сохраняет входные аргументы в локальном стеке на случай если кому-то (или чему-то) в этой функции понадобится использовать регистры  $W0 \dots W2$ , перезаписывая оригинальные аргументы функции, которые могут понадобиться в будущем. Это называется *Register Save Area*. [*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)]<sup>85</sup>. Вызываемая функция не обязана сохранять их. Это то же что и «Shadow Space»: ?? (стр. ??).

Почему оптимизирующий GCC 4.9 убрал этот, сохраняющий аргументы, код?

Потому что он провел дополнительную работу по оптимизации и сделал вывод, что аргументы функции не понадобятся в будущем и регистры  $W0 \dots W2$  также не будут использоваться.

Также мы видим пару инструкций MUL/ADD вместо одной MADD.

### 1.14.4. MIPS

Листинг 1.99: Оптимизирующий GCC 4.4.5

```
.text:00000000 f:
; $a0=a
; $a1=b
; $a2=c
.text:00000000      mult    $a1, $a0
.text:00000004      mflo   $v0
.text:00000008      jr     $ra
.text:0000000C      addu   $v0, $a2, $v0      ; branch delay slot
; результат в $v0 во время выхода
.text:00000010 main:
.text:00000010
.text:00000010 var_10 = -0x10
.text:00000010 var_4  = -4
.text:00000010
.text:00000010      lui    $gp, (__gnu_local_gp >> 16)
.text:00000014      addiu $sp, -0x20
.text:00000018      la    $gp, (__gnu_local_gp & 0xFFFF)
.text:0000001C      sw    $ra, 0x20+var_4($sp)
.text:00000020      sw    $gp, 0x20+var_10($sp)
; установить c:
.text:00000024      li    $a2, 3
; установить a:
.text:00000028      li    $a0, 1
.text:0000002C      jal   f
; установить b:
.text:00000030      li    $a1, 2      ; branch delay slot
; результат сейчас в $v0
.text:00000034      lw    $gp, 0x20+var_10($sp)
.text:00000038      lui   $a0, ($LC0 >> 16)
.text:0000003C      lw    $t9, (printf & 0xFFFF)($gp)
.text:00000040      la    $a0, ($LC0 & 0xFFFF)
.text:00000044      jalr  $t9
```

<sup>85</sup>Также доступно здесь: [http://infocenter.arm.com/help/topic/com.arm.doc.ih0055b/ih0055b\\_aapcs64.pdf](http://infocenter.arm.com/help/topic/com.arm.doc.ih0055b/ih0055b_aapcs64.pdf)

```

; взять результат ф-ции f() и передать его
; как второй аргумент в printf():
.text:00000048      move    $a1, $v0          ; branch delay slot
.text:0000004C      lw     $ra, 0x20+var_4($sp)
.text:00000050      move    $v0, $zero
.text:00000054      jr     $ra
.text:00000058      addiu  $sp, 0x20          ; branch delay slot

```

Первые 4 аргумента функции передаются в четырех регистрах с префиксами A-.

В MIPS есть два специальных регистра: HI и LO, которые выставляются в 64-битный результат умножения во время исполнения инструкции MULT.

К регистрам можно обращаться только используя инструкции MFL0 и MFHI. Здесь MFL0 берет младшую часть результата умножения и записывает в \$V0. Так что старшая 32-битная часть результата игнорируется (содержимое регистра HI не используется). Действительно, мы ведь работаем с 32-битным типом *int*.

И наконец, ADDU («Add Unsigned» — добавить беззнаковое) прибавляет значение третьего аргумента к результату.

В MIPS есть две разные инструкции сложения: ADD и ADDU. На самом деле, дело не в знаковых числах, а в исключениях: ADD может вызвать исключение во время переполнения. Это иногда полезно<sup>86</sup> и поддерживается, например, в **PL!** Ada.

ADDU не вызывает исключения во время переполнения. А так как Си/Си++ не поддерживает всё это, мы видим здесь ADDU вместо ADD.

32-битный результат оставляется в \$V0.

В `main()` есть новая для нас инструкция: JAL («Jump and Link»). Разница между JAL и JALR в том, что относительное смещение кодируется в первой инструкции, а JALR переходит по абсолютному адресу, записанному в регистр («Jump and Link Register»).

Обе функции `f()` и `main()` расположены в одном объектном файле, так что относительный адрес `f()` известен и фиксирован.

## 1.15. Ещё о возвращаемых результатах

Результат выполнения функции в x86 обычно возвращается<sup>87</sup> через регистр EAX, а если результат имеет тип байт или символ (*char*), то в самой младшей части EAX — AL. Если функция возвращает число с плавающей запятой, то будет использован регистр FPU ST(0). В ARM обычно результат возвращается в регистре R0.

<sup>86</sup><http://blog.regehr.org/archives/1154>

<sup>87</sup>См. также: MSDN: Return Values (C++): [MSDN](#)

### 1.15.1. Попытка использовать результат функции возвращающей *void*

Кстати, что будет, если возвращаемое значение в функции `main()` объявлять не как *int*, а как *void*? Т.н. `startup`-код вызывает `main()` примерно так:

```
push envp
push argv
push argc
call main
push eax
call exit
```

Иными словами:

```
exit(main(argc,argv,envp));
```

Если вы объявите `main()` как *void*, и ничего не будете возвращать явно (при помощи выражения *return*), то в единственный аргумент `exit()` попадет то, что лежало в регистре `EAX` на момент выхода из `main()`. Там, скорее всего, будет какие-то случайное число, оставшееся от работы вашей функции. Так что код завершения программы будет псевдослучайным.

Мы можем это проиллюстрировать. Заметьте, что у функции `main()` тип возвращаемого значения именно *void*:

```
#include <stdio.h>

void main()
{
    printf ("Hello, world!\n");
};
```

Скомпилируем в Linux.

GCC 4.8.1 заменила `printf()` на `puts()` (мы видели это прежде: ?? (стр. ??)), но это нормально, потому что `puts()` возвращает количество выведенных символов, так же как и `printf()`. Обратите внимание на то, что `EAX` не обнуляется перед выходом из `main()`. Это значит что `EAX` перед выходом из `main()` содержит то, что `puts()` оставляет там.

Листинг 1.100: GCC 4.8.1

```
.LC0:
    .string "Hello, world!"
main:
    push    ebp
    mov     ebp, esp
    and     esp, -16
    sub     esp, 16
    mov     DWORD PTR [esp], OFFSET FLAT:.LC0
    call   puts
    leave
    ret
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Напишем небольшой скрипт на `bash`, показывающий статус возврата («`exit status`» или «`exit code`»):

Листинг 1.101: `tst.sh`

```
#!/bin/sh
./hello_world
echo $?
```

И запустим:

```
$ tst.sh
Hello, world!
14
```

14 это как раз количество выведенных символов. Количество выведенных символов *проскальзывает* из `printf()` через `EAX/RAX` в «`exit code`».

Кстати, когда в `Hex-Rays` мы разбираем `C++` код, нередко можно наткнуться на `ф-цию`, которая заканчивается деструктором какого-либо класса:

```
...
call    ??1CString@@QAE@XZ ; CString::~CString(void)
mov     ecx, [esp+30h+var_C]
pop     edi
pop     ebx
mov     large fs:0, ecx
add     esp, 28h
retn
```

По стандарту `C++` деструкторы ничего не возвращают, но когда `Hex-Rays` об этом не знает и думает, что и деструктор и эта `ф-ция` по умолчанию возвращает `int`, то на выходе получается такой код:

```
...
        return CString::~CString(&Str);
}
```

### 1.15.2. Что если не использовать результат функции?

`printf()` возвращает количество успешно выведенных символов, но результат работы этой функции редко используется на практике.

Можно даже явно вызывать функции, чей смысл именно в возвращаемых значениях, но явно не использовать их:

```
int f()
{
    // пропускаем первые 3 случайных значения:
    rand();
    rand();
}
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    rand();
    // и используем 4-е:
    return rand();
};

```

Результат работы `rand()` остается в EAX во всех четырех случаях. Но в первых трех случаях значение, лежащее в EAX, просто не используется.

### 1.15.3. Возврат структуры

Вернемся к тому факту, что возвращаемое значение остается в регистре EAX. Вот почему старые компиляторы Си не способны создавать функции, возвращающие нечто большее, нежели помещается в один регистр (обычно тип *int*), а когда нужно, приходится возвращать через указатели, указываемые в аргументах. Так что, как правило, если функция должна вернуть несколько значений, она возвращает только одно, а остальные — через указатели. Хотя позже и стало возможным, вернуть, скажем, целую структуру, но этот метод до сих пор не очень популярен. Если функция должна вернуть структуру, вызывающая функция должна сама, скрыто и прозрачно для программиста, выделить место и передать указатель на него в качестве первого аргумента. Это почти то же самое что и сделать это вручную, но компилятор прячет это.

Небольшой пример:

```

struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    struct s rt;

    rt.a=a+1;
    rt.b=a+2;
    rt.c=a+3;

    return rt;
};

```

...получим (MSVC 2010 /0x):

```

$T3853 = 8                ; size = 4
_a$ = 12                  ; size = 4
?get_some_values@@YA?AU s@@H@Z PROC          ; get_some_values
    mov     ecx, DWORD PTR _a$[esp-4]
    mov     eax, DWORD PTR $T3853[esp-4]
    lea    edx, DWORD PTR [ecx+1]
    mov     DWORD PTR [eax], edx
    lea    edx, DWORD PTR [ecx+2]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

add    ecx, 3
mov    DWORD PTR [eax+4], edx
mov    DWORD PTR [eax+8], ecx
ret    0
?get_some_values@@YA?AU?AU@H@Z ENDP          ; get_some_values

```

\$T3853 это имя внутреннего макроса для передачи указателя на структуру. Этот пример можно даже переписать, используя расширения C99:

```

struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    return (struct s){.a=a+1, .b=a+2, .c=a+3};
};

```

Листинг 1.102: GCC 4.8.1

```

_get_some_values proc near
ptr_to_struct  = dword ptr  4
a              = dword ptr  8

        mov     edx, [esp+a]
        mov     eax, [esp+ptr_to_struct]
        lea    ecx, [edx+1]
        mov     [eax], ecx
        lea    ecx, [edx+2]
        add    edx, 3
        mov     [eax+4], ecx
        mov     [eax+8], edx
        retn
_get_some_values endp

```

Как видно, функция просто заполняет поля в структуре, выделенной вызывающей функцией. Как если бы передавался просто указатель на структуру. Так что никаких проблем с эффективностью нет.

## 1.16. Указатели

### 1.16.1. Возврат значений

Указатели также часто используются для возврата значений из функции (вспомните случай со `scanf()` (`??` (стр. `??`))).

Например, когда функции нужно вернуть сразу два значения.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

**Пример с глобальными переменными**

```
#include <stdio.h>

void f1 (int x, int y, int *sum, int *product)
{
    *sum=x+y;
    *product=x*y;
};

int sum, product;

void main()
{
    f1(123, 456, &sum, &product);
    printf ("sum=%d, product=%d\n", sum, product);
};
```

Это компилируется в:

Листинг 1.103: Оптимизирующий MSVC 2010 (/Ob0)

```
COMM    _product:DWORD
COMM    _sum:DWORD
$SG2803 DB      'sum=%d, product=%d', 0aH, 00H

_x$ = 8          ; size = 4
_y$ = 12         ; size = 4
_sum$ = 16       ; size = 4
_product$ = 20   ; size = 4
_f1      PROC
    mov     ecx, DWORD PTR _y$[esp-4]
    mov     eax, DWORD PTR _x$[esp-4]
    lea    edx, DWORD PTR [eax+ecx]
    imul   eax, ecx
    mov     ecx, DWORD PTR _product$[esp-4]
    push   esi
    mov     esi, DWORD PTR _sum$[esp]
    mov     DWORD PTR [esi], edx
    mov     DWORD PTR [ecx], eax
    pop    esi
    ret    0
_f1      ENDP

_main    PROC
    push   OFFSET _product
    push   OFFSET _sum
    push   456      ; 000001c8H
    push   123     ; 0000007bH
    call  _f1
    mov    eax, DWORD PTR _product
    mov    ecx, DWORD PTR _sum
    push  eax
    push  ecx
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```
    push    OFFSET $SG2803
    call    DWORD PTR __imp__printf
    add     esp, 28
    xor     eax, eax
    ret     0
_main    ENDP
```

Посмотрим это в OllyDbg:

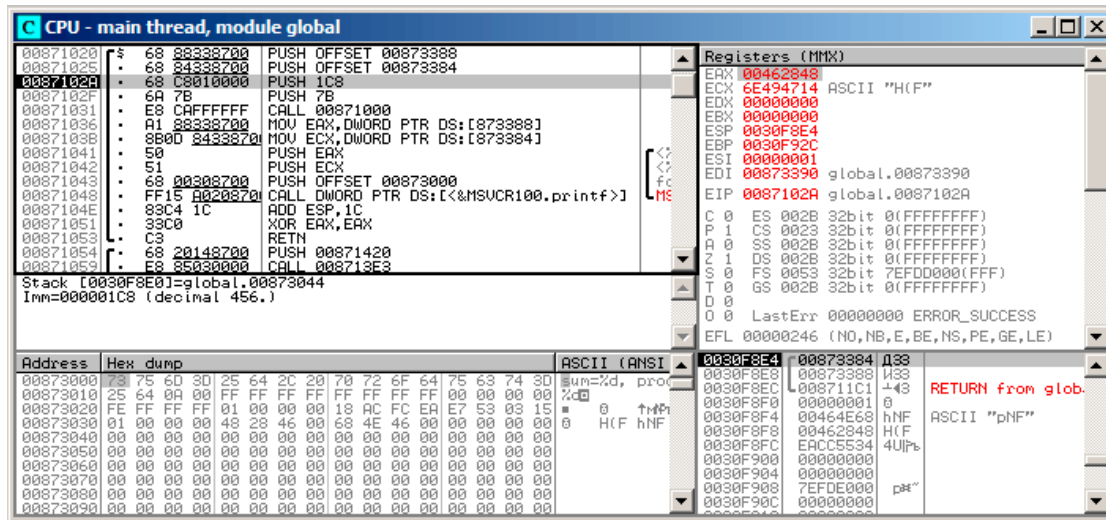


Рис. 1.24: OllyDbg: передаются адреса двух глобальных переменных в `f1()`

В начале адреса обоих глобальных переменных передаются в `f1()`. Можно нажать «Follow in dump» на элементе стека и в окне слева увидим место в сегменте данных, выделенное для двух переменных.

Эти переменные обнулены, потому что по стандарту неинициализированные данные (**BSS!**) обнуляются перед началом исполнения: [ISO/IEC 9899:TC3 (C99 standard), (2007)6.7.8p10].

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

И они находятся в сегменте данных, о чем можно удостовериться, нажав Alt-M и увидев карту памяти:

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00050000	00004000				Map	R	R	
00060000	00001000				Priv	RW	RW	
00070000	00007000				Map	R	R	C:\Windows\System32\Loc
00159000	00007000				Priv	RW GUA	RW GUA	
00300000	00001000				Priv	RW GUA	RW GUA	
0030E000	00002000			Stack of main thread	Priv	RW	RW	
00460000	00005000			Heap	Priv	RW	RW	
004A0000	00007000				Priv	RW	RW	
00660000	0000C000			Default heap	Priv	RW	RW	
00870000	00001000	global		PE header	Img	R	RWE	Copt
00871000	00001000	global	.text	Code	Img	R E	RWE	Copt
00872000	00001000	global	.rdata	Imports	Img	R	RWE	Copt
00873000	00001000	global	.data	Data	Img	RW	RWE	Copt
00874000	00001000	global	.reloc	Relocations	Img	R	RWE	Copt
6E3E0000	00001000	MSUCR100		PE header	Img	R	RWE	Copt
6E3E1000	00002000	MSUCR100	.text	Code, imports, exports	Img	R E	RWE	Copt
6E493000	00006000	MSUCR100	.data	Data	Img	RW	RWE	Copt
6E499000	00001000	MSUCR100	.rsrc	Resources	Img	R	RWE	Copt
6E49A000	00005000	MSUCR100	.reloc	Relocations	Img	R	RWE	Copt
755D0000	00001000	Mod_755D		PE header	Img	R	RWE	Copt
755D1000	00003000				Img	R E	RWE	Copt
755D4000	00001000				Img	RW	RWE	Copt
755D5000	00003000				Img	R	RWE	Copt
755E0000	00001000	Mod_755E		PE header	Img	R	RWE	Copt
755E1000	00004000				Img	R E	RWE	Copt
7562E000	00005000				Img	RW	RWE	Copt
75633000	00009000				Img	R	RWE	Copt

Рис. 1.25: OllyDbg: карта памяти

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мой адреса](#). Спасибо!

Трассируем (F7) до начала исполнения f1():

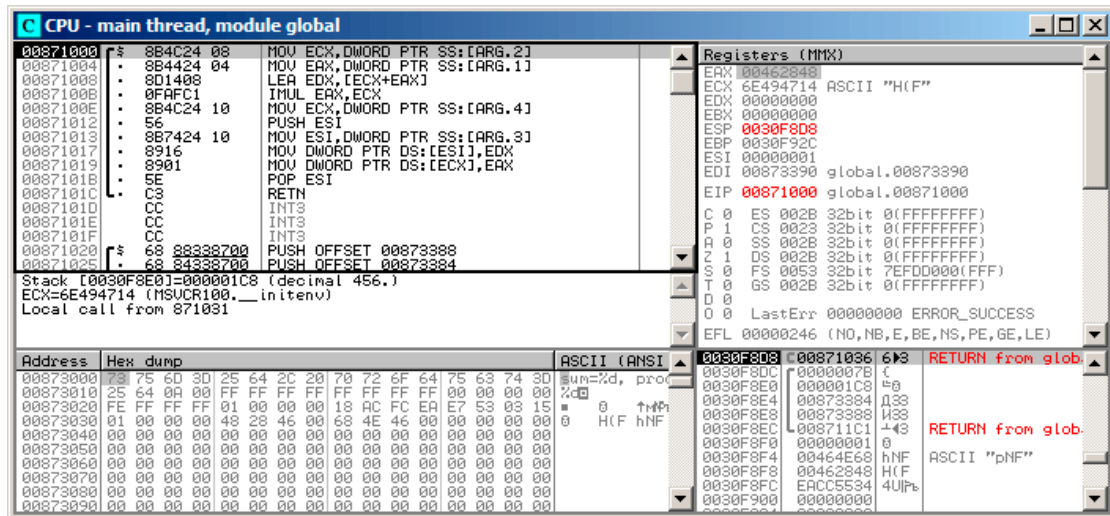


Рис. 1.26: OllyDbg: начало работы f1()

В стеке видны значения 456 (0x1C8) и 123 (0x7B), а также адреса двух глобальных переменных.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Трассируем до конца `f1()`. Мы видим в окне слева, как результаты вычисления появились в глобальных переменных:

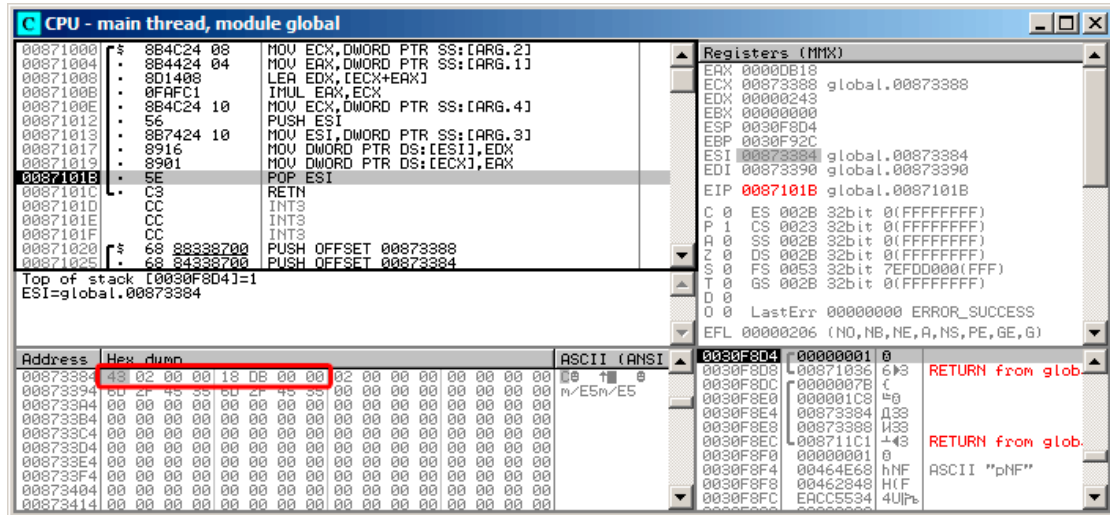


Рис. 1.27: OllyDbg: `f1()` заканчивает работу

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Теперь из глобальных переменных значения загружаются в регистры для передачи в printf():

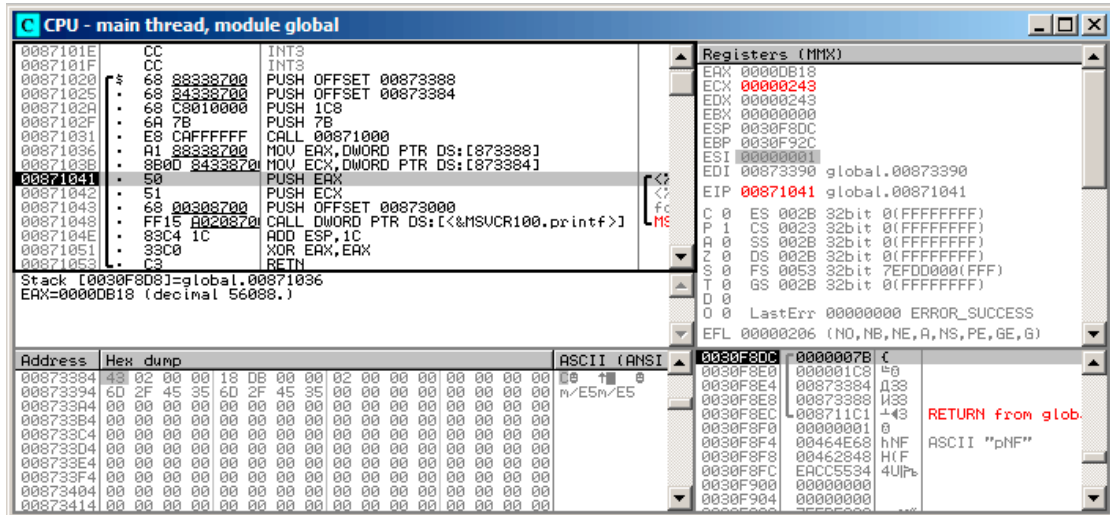


Рис. 1.28: OllyDbg: адреса глобальных переменных передаются в printf()

### Пример с локальными переменными

Немного переделаем пример:

Листинг 1.104: теперь переменные локальные

```

void main()
{
    int sum, product; // теперь эти переменные для этой ф-ции ---
    локальные

    f1(123, 456, &sum, &product);
    printf ("sum=%d, product=%d\n", sum, product);
};
  
```

Код функции f1() не изменится. Изменится только main():

Листинг 1.105: Оптимизирующий MSVC 2010 (/Ob0)

```

_product$ = -8 ; size = 4
_sum$ = -4 ; size = 4
_main PROC
; Line 10
sub esp, 8
; Line 13
lea eax, DWORD PTR _product$[esp+8]
push eax
lea ecx, DWORD PTR _sum$[esp+12]
push ecx
  
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
    push    456      ; 000001c8H
    push    123      ; 0000007bH
    call    _f1
; Line 14
    mov     edx, DWORD PTR _product$[esp+24]
    mov     eax, DWORD PTR _sum$[esp+24]
    push    edx
    push    eax
    push    OFFSET $SG2803
    call    DWORD PTR __imp__printf
; Line 15
    xor     eax, eax
    add     esp, 36
    ret     0
```

Снова посмотрим в OllyDbg. Адреса локальных переменных в стеке это 0x2EF854 и 0x2EF858. Видно, как они заталкиваются в стек:

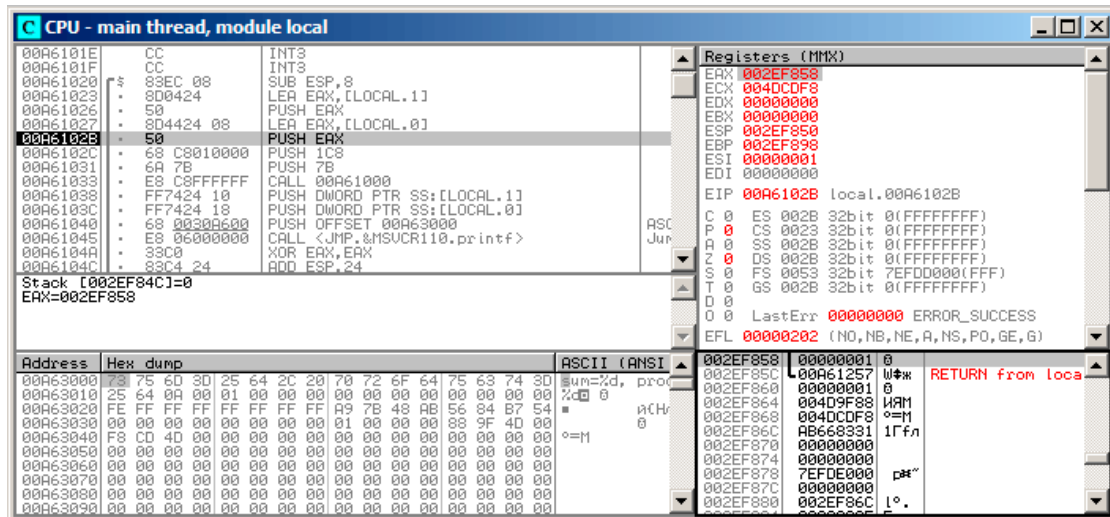


Рис. 1.29: OllyDbg: адреса локальных переменных заталкиваются в стек

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



Начало работы `f1()`. В стеке по адресам `0x2EF854` и `0x2EF858` пока находится случайный мусор:

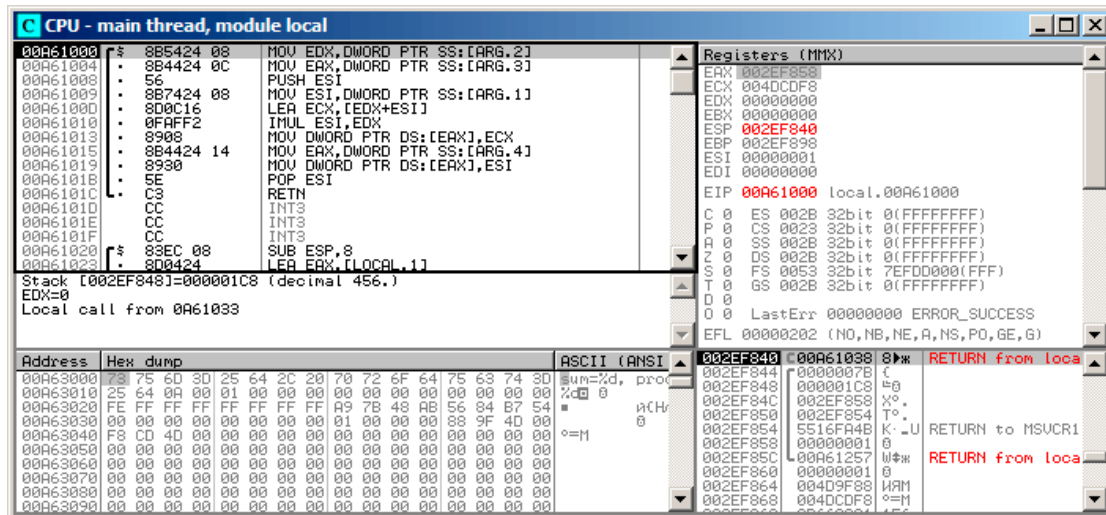


Рис. 1.30: OllyDbg: `f1()` начинает работу

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Конец работы f1():

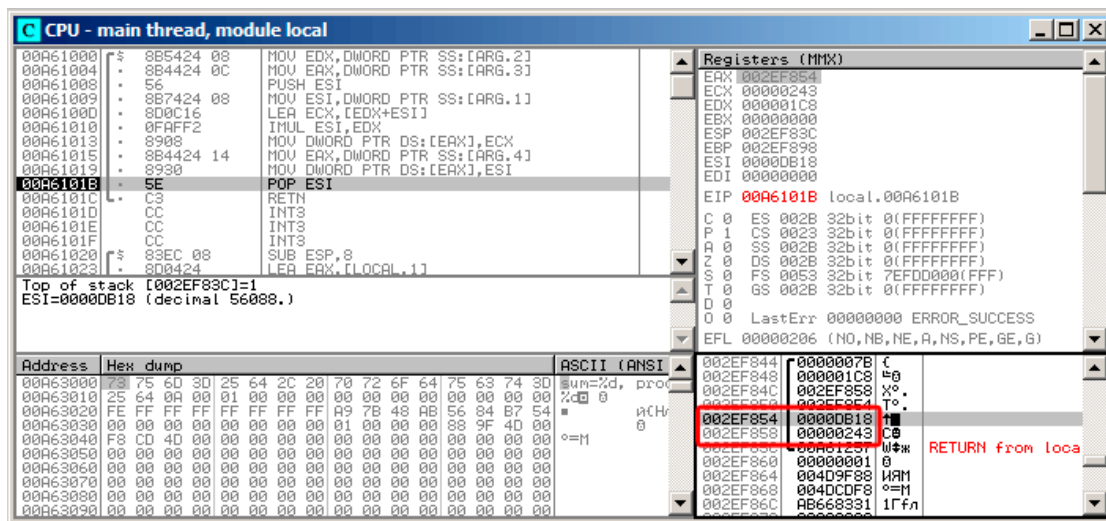


Рис. 1.31: OllyDbg: f1() заканчивает работу

В стеке по адресам 0x2EF854 и 0x2EF858 теперь находятся значения 0xDB18 и 0x243, это результаты работы f1().

## Вывод

f1() может одинаково хорошо возвращать результаты работы в любые места памяти. В этом суть и удобство указателей. Кстати, *references* в Си++ работают точно так же. Читайте больше об этом: (?? (стр. ??)).

## 1.16.2. Обменять входные значения друг с другом

Вот так:

```
#include <memory.h>
#include <stdio.h>

void swap_bytes (unsigned char* first, unsigned char* second)
{
    unsigned char tmp1;
    unsigned char tmp2;

    tmp1=*first;
    tmp2=*second;

    *first=tmp2;
    *second=tmp1;
};
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

int main()
{
    // копируем строку в кучу, чтобы у нас была возможность эту самую
    // строку модифицировать
    char *s=strdup("string");

    // меняем 2-й и 3-й символы
    swap_bytes (s+1, s+2);

    printf ("%s\n", s);
};

```

Как видим, байты загружаются в младшие 8-битные части регистров ECX и EBX используя MOVZX (так что старшие части регистров очищаются), затем байты записываются назад в другом порядке.

Листинг 1.106: Optimizing GCC 5.4

```

swap_bytes:
    push    ebx
    mov     edx, DWORD PTR [esp+8]
    mov     eax, DWORD PTR [esp+12]
    movzx  ecx, BYTE PTR [edx]
    movzx  ebx, BYTE PTR [eax]
    mov     BYTE PTR [edx], bl
    mov     BYTE PTR [eax], cl
    pop     ebx
    ret

```

Адреса обоих байтов берутся из аргументов и во время исполнения ф-ции находятся в регистрах EDX и EAX.

Так что используем указатели — вероятно, без них нет способа решить эту задачу лучше.

## 1.17. Оператор GOTO

Оператор GOTO считается анти-паттерном, см: [Edgar Dijkstra, *Go To Statement Considered Harmful* (1968)<sup>88</sup>]. Но тем не менее, его можно использовать в разумных пределах, см: [Donald E. Knuth, *Structured Programming with go to Statements* (1974)<sup>89</sup> <sup>90</sup>.

Вот простейший пример:

```

#include <stdio.h>

int main()
{

```

<sup>88</sup><http://yurichev.com/mirrors/Dijkstra68.pdf>

<sup>89</sup><http://yurichev.com/mirrors/KnuthStructuredProgrammingGoTo.pdf>

<sup>90</sup>В [Денис Юричев, *Заметки о языке программирования Си/Си++*] также есть примеры.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    printf ("begin\n");
    goto exit;
    printf ("skip me!\n");
exit:
    printf ("end\n");
};

```

Вот что мы получаем в MSVC 2012:

Листинг 1.107: MSVC 2012

```

$SG2934 DB      'begin', 0aH, 00H
$SG2936 DB      'skip me!', 0aH, 00H
$SG2937 DB      'end', 0aH, 00H

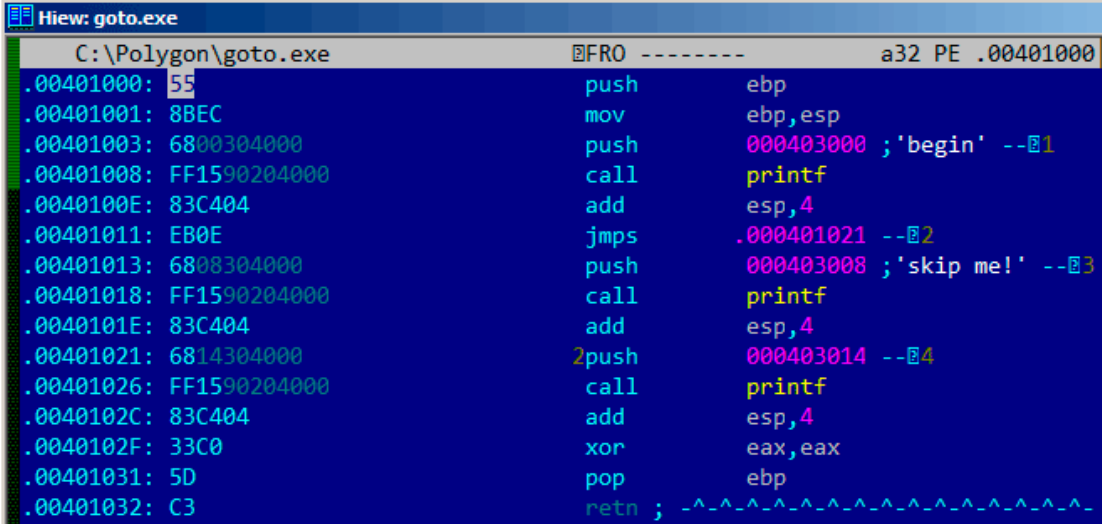
_main  PROC
        push    ebp
        mov     ebp, esp
        push   OFFSET $SG2934 ; 'begin'
        call   _printf
        add    esp, 4
        jmp    SHORT $exit$3
        push   OFFSET $SG2936 ; 'skip me!'
        call   _printf
        add    esp, 4
$exit$3:
        push   OFFSET $SG2937 ; 'end'
        call   _printf
        add    esp, 4
        xor    eax, eax
        pop    ebp
        ret    0
_main  ENDP

```

Выражение *goto* заменяется инструкцией JMP, которая работает точно также: безусловный переход в другое место. Вызов второго printf() может исполниться только при помощи человеческого вмешательства, используя отладчик или модифицирование кода.

Это также может быть простым упражнением на модификацию кода.

Откроем исполняемый файл в Hiew:



```
Hiew: goto.exe
C:\Polygon\goto.exe  FRO ----- a32 PE .00401000
.00401000: 55      push  ebp
.00401001: 8BEC    mov   ebp,esp
.00401003: 6800304000 push 000403000 ;'begin' --01
.00401008: FF1590204000 call printf
.0040100E: 83C404  add  esp,4
.00401011: EB0E    jmps .000401021 --02
.00401013: 6800304000 push 000403008 ;'skip me!' --03
.00401018: FF1590204000 call printf
.0040101E: 83C404  add  esp,4
.00401021: 6814304000 2push 000403014 --04
.00401026: FF1590204000 call printf
.0040102C: 83C404  add  esp,4
.0040102F: 33C0    xor   eax,eax
.00401031: 5D      pop   ebp
.00401032: C3      retn ; _^_ _^_ _^_ _^_ _^_ _^_ _^_ _^_ _^_
```

Рис. 1.32: Hiew

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Поместите курсор по адресу JMP (0x410), нажмите F3 (редактирование), нажмите два нуля, так что опкод становится EB 00:

```

Hiew: goto.exe
C:\Polygon\goto.exe  FWO EDITMODE  a32 PE  00000413
00000400: 55          push     ebp
00000401: 8BEC       mov     ebp,esp
00000403: 6800304000 push    000403000 ;' @0 '
00000408: FF1590204000 call   d,[000402090]
0000040E: 83C404     add     esp,4
00000411: EB00      jmps   00000413
00000413: 6800304000 push    000403008 ;' @00 '
00000418: FF1590204000 call   d,[000402090]
0000041E: 83C404     add     esp,4
00000421: 6814304000 push    000403014 ;' @000 '
00000426: FF1590204000 call   d,[000402090]
0000042C: 83C404     add     esp,4
0000042F: 33C0      xor     eax,eax
00000431: 5D        pop     ebp
00000432: C3       retn   ; ^^^^_^^^_^^^_^^^_^^^_^^^_^^^_^^^_^^^_^^^_^^^_^^^

```

Рис. 1.33: Hiew

Второй байт опкода JMP это относительное смещение от перехода. 0 означает место прямо после текущей инструкции. Теперь JMP не будет пропускать следующий вызов printf(). Нажмите F9 (запись) и выйдите. Теперь мы запускаем исполняемый файл и видим это:

Листинг 1.108: Результат

```

C:\...\>goto.exe

begin
skip me!
end

```

Подобного же эффекта можно достичь, если заменить инструкцию JMP на две инструкции NOP. NOP имеет опкод 0x90 и длину в 1 байт, так что нужно 2 инструкции для замены.

### 1.17.1. Мертвый код

Вызов второго printf() также называется «мертвым кодом» («dead code») в терминах компиляторов. Это значит, что он никогда не будет исполнен. Так что если вы компилируете этот пример с оптимизацией, компилятор удаляет «мертвый код» не оставляя следа:

Листинг 1.109: Оптимизирующий MSVC 2012

```

$SG2981 DB 'begin', 0aH, 00H

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

$SG2983 DB      'skip me!', 0aH, 00H
$SG2984 DB      'end', 0aH, 00H

_main  PROC
        push    OFFSET $SG2981 ; 'begin'
        call    _printf
        push    OFFSET $SG2984 ; 'end'
$exit$4:
        call    _printf
        add     esp, 8
        xor     eax, eax
        ret     0
_main  ENDP

```

Впрочем, строку «skip me!» компилятор убрать забыл.

### 1.17.2. Упражнение

Попробуйте добиться того же самого в вашем любимом компиляторе и отладчике.

## 1.18. Условные переходы

### 1.18.1. Простой пример

```

#include <stdio.h>

void f_signed (int a, int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

void f_unsigned (unsigned int a, unsigned int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

int main()
{
    f_signed(1, 2);
    f_unsigned(1, 2);
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
    return 0;
};
```

## x86

### x86 + MSVC

Имеем в итоге функцию `f_signed()`:

Листинг 1.110: Неоптимизирующий MSVC 2010

```
_a$ = 8
_b$ = 12
_f_signed PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    cmp     eax, DWORD PTR _b$[ebp]
    jle     SHORT $LN3@f_signed
    push    OFFSET $SG737          ; 'a>b'
    call   _printf
    add     esp, 4
$LN3@f_signed:
    mov     ecx, DWORD PTR _a$[ebp]
    cmp     ecx, DWORD PTR _b$[ebp]
    jne     SHORT $LN2@f_signed
    push    OFFSET $SG739          ; 'a==b'
    call   _printf
    add     esp, 4
$LN2@f_signed:
    mov     edx, DWORD PTR _a$[ebp]
    cmp     edx, DWORD PTR _b$[ebp]
    jge     SHORT $LN4@f_signed
    push    OFFSET $SG741          ; 'a<b'
    call   _printf
    add     esp, 4
$LN4@f_signed:
    pop     ebp
    ret     0
_f_signed ENDP
```

Первая инструкция `JLE` значит *Jump if Less or Equal*. Если второй операнд больше первого или равен ему, произойдет переход туда, где будет следующая проверка.

А если это условие не срабатывает (то есть второй операнд меньше первого), то перехода не будет, и сработает первый `printf()`.

Вторая проверка это `JNE`: *Jump if Not Equal*. Переход не произойдет, если операнды равны.

Третья проверка `JGE`: *Jump if Greater or Equal* — переход если первый операнд больше второго или равен ему. Кстати, если все три условных перехода сра-

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



ботаю, ни один `printf()` не вызовется. Но без внешнего вмешательства это невозможно.

Функция `f_unsigned()` точно такая же, за тем исключением, что используются инструкции `JBE` и `JAЕ` вместо `JLE` и `JGE`:

Листинг 1.111: GCC

```

_a$ = 8 ; size = 4
_b$ = 12 ; size = 4
_f_unsigned PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    cmp     eax, DWORD PTR _b$[ebp]
    jbe     SHORT $LN3@f_unsigned
    push    OFFSET $SG2761 ; 'a>b'
    call   _printf
    add     esp, 4
$LN3@f_unsigned:
    mov     ecx, DWORD PTR _a$[ebp]
    cmp     ecx, DWORD PTR _b$[ebp]
    jne     SHORT $LN2@f_unsigned
    push    OFFSET $SG2763 ; 'a==b'
    call   _printf
    add     esp, 4
$LN2@f_unsigned:
    mov     edx, DWORD PTR _a$[ebp]
    cmp     edx, DWORD PTR _b$[ebp]
    jae     SHORT $LN4@f_unsigned
    push    OFFSET $SG2765 ; 'a<b'
    call   _printf
    add     esp, 4
$LN4@f_unsigned:
    pop     ebp
    ret     0
_f_unsigned ENDP

```

Здесь всё то же самое, только инструкции условных переходов немного другие:

`JBE`— *Jump if Below or Equal* и `JAЕ`— *Jump if Above or Equal*. Эти инструкции (`JA/JAE/JB/JBE`) отличаются от `JG/JGE/JL/JLE` тем, что работают с беззнаковыми переменными.

Таким образом, увидев где используется `JG/JL` вместо `JA/JB` и наоборот, можно сказать почти уверенно насчет того, является ли тип переменной знаковым (`signed`) или беззнаковым (`unsigned`).

Далее функция `main()`, где ничего нового для нас нет:

Листинг 1.112: `main()`

```

_main    PROC
        push    ebp

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
    mov     ebp, esp
    push   2
    push   1
    call   _f_signed
    add    esp, 8
    push   2
    push   1
    call   _f_unsigned
    add    esp, 8
    xor    eax, eax
    pop    ebp
    ret    0
_main   ENDP
```

## x86 + MSVC + OllyDbg

Если попробовать этот пример в OllyDbg, можно увидеть, как выставляются флаги. Начнем с функции `f_unsigned()`, которая работает с беззнаковыми числами.

В целом в каждой функции `cmp` исполняется три раза, но для одних и тех же аргументов, так что флаги все три раза будут одинаковы.

Результат первого сравнения:

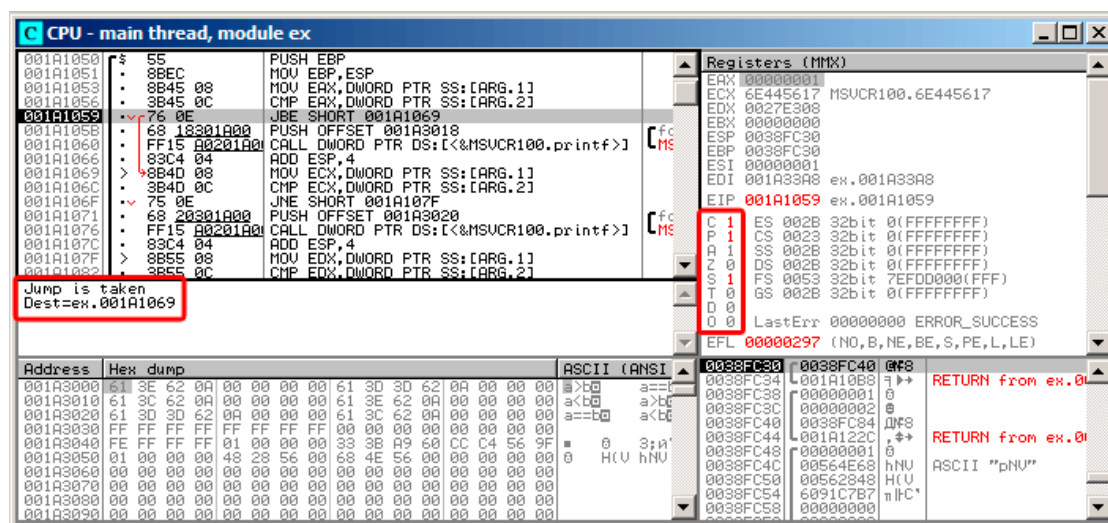


Рис. 1.34: OllyDbg: `f_unsigned()`: первый условный переход

Итак, флаги: `C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0`. Для краткости, в OllyDbg флаги называются только одной буквой.

OllyDbg подсказывает, что первый переход (`JBE`) сейчас сработает. Действительно, если заглянуть в документацию от Intel, (`??` (стр. `??`)) прочитаем там, что `JBE` срабатывает в случаях если `CF=1` или `ZF=1`. Условие здесь выполняется, так что переход срабатывает.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Следующий переход:

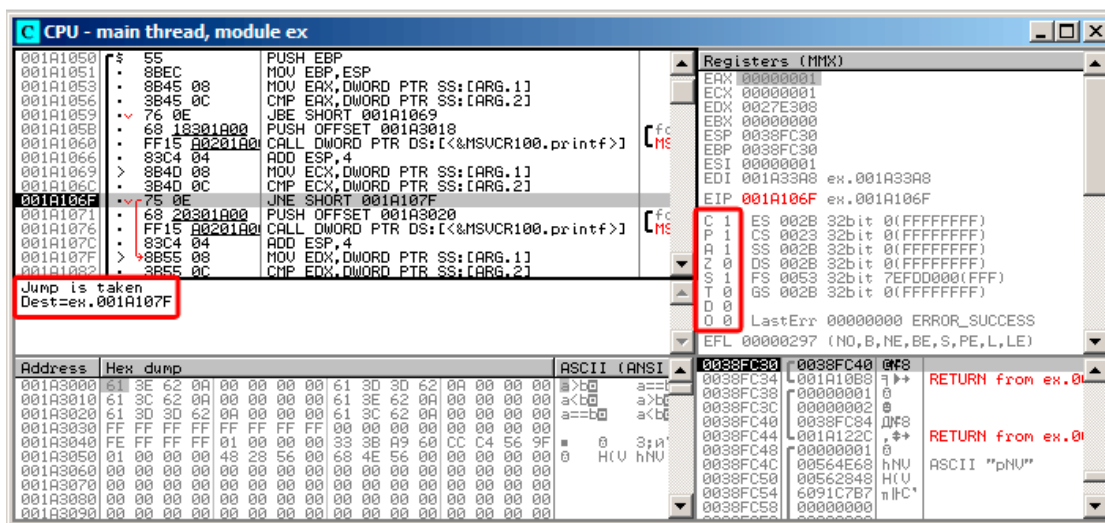


Рис. 1.35: OllyDbg: f\_unsigned(): второй условный переход

OllyDbg подсказывает, что JNZ сработает. Действительно, JNZ срабатывает если ZF=0 (zero flag).

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Третий переход, JNB:

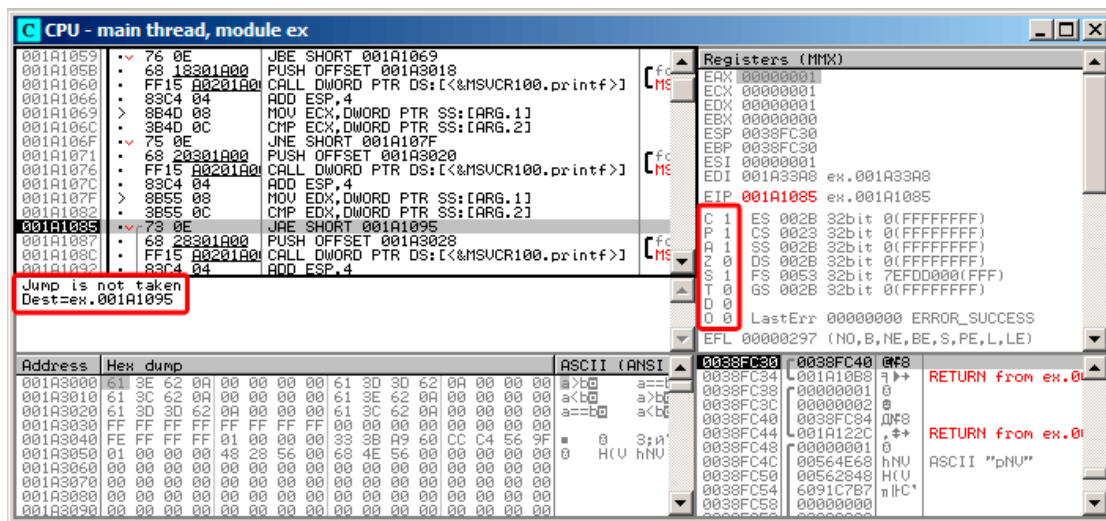


Рис. 1.36: OllyDbg: `f_unsigned()`: третий условный переход

В документации от Intel (?? (стр. ??)) мы можем найти, что JNB срабатывает если  $CF=0$  (carry flag). В нашем случае это не так, переход не срабатывает, и исполняется третий по счету `printf()`.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Теперь можно попробовать в OllyDbg функцию `f_signed()`, работающую со знаковыми величинами. Флаги выставляются точно так же: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0. Первый переход JLE работает:

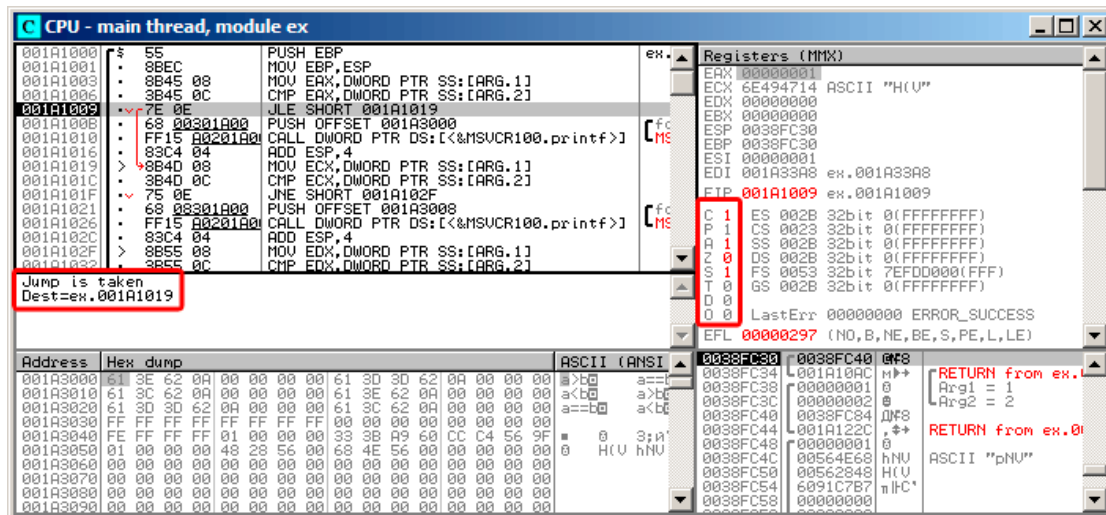


Рис. 1.37: OllyDbg: `f_signed()`: первый условный переход

В документации от Intel (?? (стр. ??)) мы можем прочитать, что эта инструкция срабатывает если  $ZF=1$  или  $SF \neq OF$ . В нашем случае  $SF \neq OF$ , так что переход срабатывает.

Второй переход JNZ работает: он срабатывает если ZF=0 (zero flag):

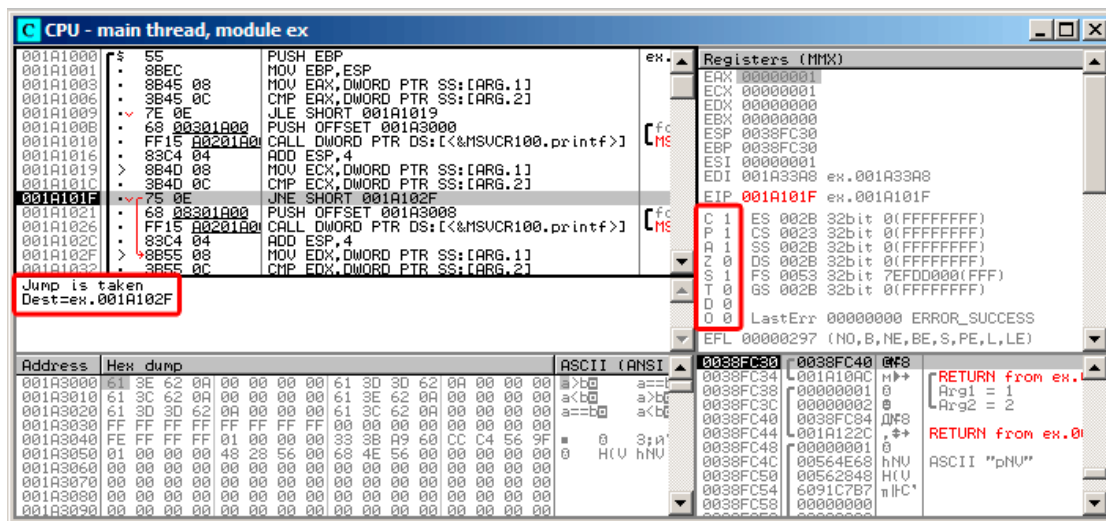


Рис. 1.38: OllyDbg: f\_signed(): второй условный переход

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Третий переход JGE не сработает, потому что он срабатывает, только если SF=OF, что в нашем случае не так:

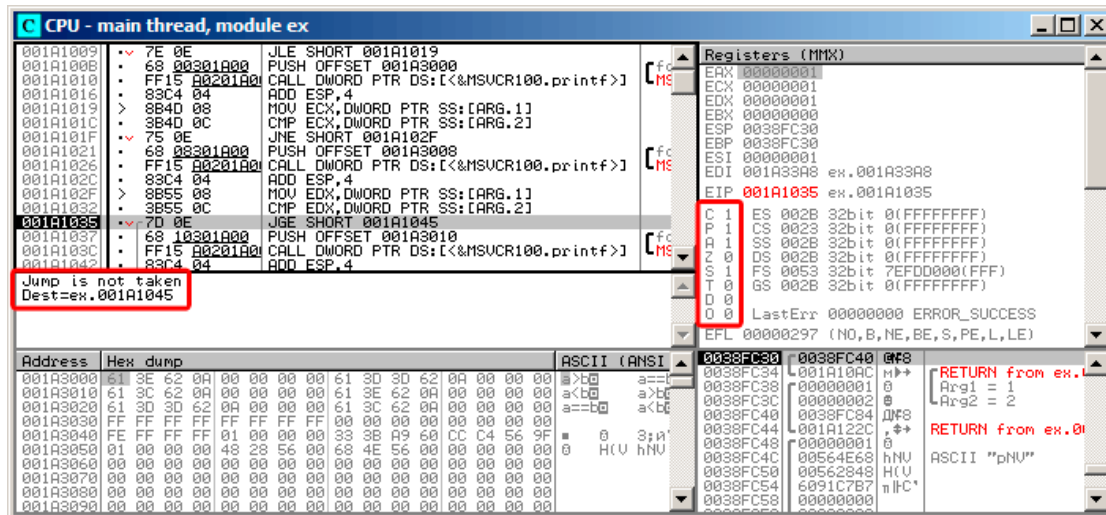


Рис. 1.39: OllyDbg: f\_signed(): третий условный переход

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



## x86 + MSVC + Hiew

Можем попробовать модифицировать исполняемый файл так, чтобы функция `f_unsigned()` всегда показывала «a==b», при любых входящих значениях. Вот как она выглядит в Hiew:

```
C:\Polygon\ollydbg\7_1.exe      FRO -----      a32 PE .00401000  Hiew 8.02 (c)SEN
.00401000: 55      push     ebp
.00401001: 8BEC    mov     ebp, esp
.00401003: 8B4508  mov     eax, [ebp+8]
.00401006: 3B450C  cmp     eax, [ebp+0C]
.00401009: 7E0D    jle     .00401018 --E1
.0040100B: 6800B04000  push   00040B000 --E2
.00401010: E8AA000000  call   .004010BF --E3
.00401015: 83C404  add     esp, 4
.00401018: 8B4D08  1mov   ecx, [ebp+8]
.0040101B: 3B4D0C  cmp     ecx, [ebp+0C]
.0040101E: 750D    jnz     .0040102D --E4
.00401020: 6800B04000  push   00040B008 ; 'a==b' --E5
.00401025: E895000000  call   .004010BF --E3
.0040102A: 83C404  add     esp, 4
.0040102D: 8B5508  4mov   edx, [ebp+8]
.00401030: 3B550C  cmp     edx, [ebp+0C]
.00401033: 7D0D    jge     .00401042 --E6
.00401035: 6810B04000  push   00040B010 --E7
.0040103A: E880000000  call   .004010BF --E3
.0040103F: 83C404  add     esp, 4
.00401042: 5D      6pop   ebp
.00401043: C3      ret;
.00401044: CC      int    3
.00401045: CC      int    3
.00401046: CC      int    3
.00401047: CC      int    3
.00401048: CC      int    3
1Global 2FillBlk 3CryBlk 4ReLoad 5OrdLdr 6String 7Direct 8Table 91byte 10Leave 11Naked 12AddNam
```

Рис. 1.40: Hiew: функция `f_unsigned()`

Собственно, задач три:

- заставить первый переход срабатывать всегда;
- заставить второй переход не срабатывать никогда;
- заставить третий переход срабатывать всегда.

Так мы направим путь исполнения кода (code flow) во второй `printf()`, и он всегда будет срабатывать и выводить на консоль «a==b».

Для этого нужно изменить три инструкции (или байта):

- Первый переход теперь будет `JMP`, но смещение перехода (`jump offset`) останется прежним.
- Второй переход может быть и будет срабатывать иногда, но в любом случае он будет совершать переход только на следующую инструкцию, пото-

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

---

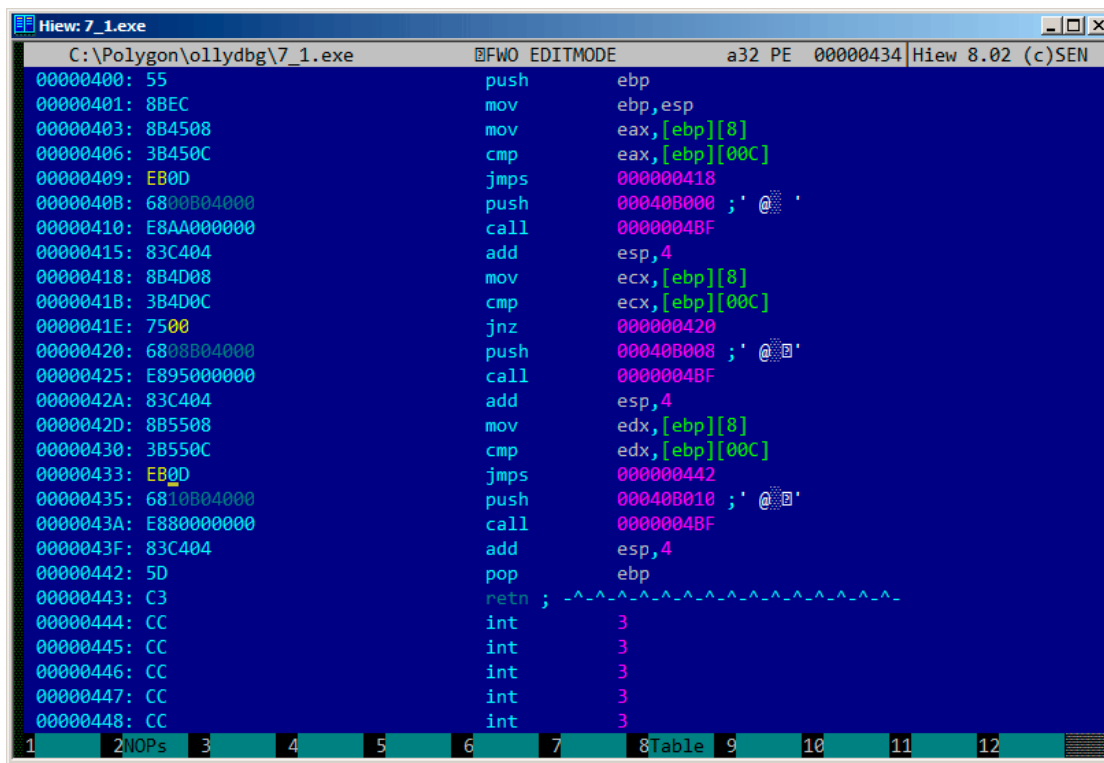
му что мы выставяем смещение перехода ([jump offset](#)) в 0.

В этих инструкциях смещение перехода просто прибавляется к адресу следующей инструкции.

Когда смещение 0, переход будет на следующую инструкцию.

- Третий переход конвертируем в JMP точно так же, как и первый, он будет срабатывать всегда.

Что и делаем:



```

C:\Polygon\ollydbg\7_1.exe  FWO EDITMODE  a32 PE  00000434 Hiew 8.02 (c)SEN
00000400: 55      push   ebp
00000401: 8BEC   mov    ebp,esp
00000403: 8B4508 mov    eax,[ebp+8]
00000406: 3B450C cmp    eax,[ebp+00C]
00000409: EB0D   jmps   00000418
0000040B: 6800B04000 push  00040B00 ; '@'
00000410: E8AA000000 call   000004BF
00000415: 83C404 add    esp,4
00000418: 8B4D08 mov    ecx,[ebp+8]
0000041B: 3B4D0C cmp    ecx,[ebp+00C]
0000041E: 750B   jnz    00000420
00000420: 6800B04000 push  00040B00 ; '@'
00000425: E895000000 call   000004BF
0000042A: 83C404 add    esp,4
0000042D: 8B5508 mov    edx,[ebp+8]
00000430: 3B550C cmp    edx,[ebp+00C]
00000433: EB0D   jmps   00000442
00000435: 6810B04000 push  00040B10 ; '@'
0000043A: E880000000 call   000004BF
0000043F: 83C404 add    esp,4
00000442: 5D     pop    ebp
00000443: C3     retn  ; ~^^^~^~^~^~^~^~^~^~^~^~^~^
00000444: CC     int    3
00000445: CC     int    3
00000446: CC     int    3
00000447: CC     int    3
00000448: CC     int    3
1      2 NOPS  3      4      5      6      7      8 Table  9      10     11     12

```

Рис. 1.41: Hiew: модифицируем функцию `f_unsigned()`

Если забыть про какой-то из переходов, то тогда будет срабатывать несколько вызовов `printf()`, а нам ведь нужно чтобы исполнялся только один.

### Неоптимизирующий GCC

Неоптимизирующий GCC 4.4.1 производит почти такой же код, за исключением `puts()` (`??` (стр. `??`)) вместо `printf()`.

### Оптимизирующий GCC

Наблюдательный читатель может спросить, зачем исполнять `cmp` так много раз, если флаги всегда одни и те же? По-видимому, оптимизирующий `MSVC` не может этого делать, но `GCC 4.8.1` делает больше оптимизаций:

Листинг 1.113: GCC 4.8.1 `f_signed()`

```

f_signed:
    mov    eax, DWORD PTR [esp+8]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        cmp     DWORD PTR [esp+4], eax
        jg     .L6
        je     .L7
        jge   .L1
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC2 ; "a<b"
        jmp    puts
.L6:
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0 ; "a>b"
        jmp    puts
.L1:
        rep   ret
.L7:
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC1 ; "a==b"
        jmp    puts

```

Мы здесь также видим JMP puts вместо CALL puts / RETN. Этот прием описан немного позже: ?? (стр. ??).

Нужно сказать, что x86-код такого типа редок. MSVC 2012, как видно, не может генерировать подобное. С другой стороны, программисты на ассемблере прекрасно осведомлены о том, что инструкции Jcc можно располагать последовательно.

Так что если вы видите это где-то, имеется немалая вероятность, что этот фрагмент кода был написан вручную.

Функция f\_unsigned() получилась не настолько эстетически короткой:

Листинг 1.114: GCC 4.8.1 f\_unsigned()

```

f_unsigned:
    push     esi
    push     ebx
    sub     esp, 20
    mov     esi, DWORD PTR [esp+32]
    mov     ebx, DWORD PTR [esp+36]
    cmp     esi, ebx
    ja     .L13
    cmp     esi, ebx ; эту инструкцию можно было бы убрать
    je     .L14
.L10:
    jb     .L15
    add     esp, 20
    pop     ebx
    pop     esi
    ret
.L15:
    mov     DWORD PTR [esp+32], OFFSET FLAT:.LC2 ; "a<b"
    add     esp, 20
    pop     ebx
    pop     esi
    jmp    puts
.L13:
    mov     DWORD PTR [esp], OFFSET FLAT:.LC0 ; "a>b"
    call   puts

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        cmp     esi, ebx
        jne     .L10
.L14:
        mov     DWORD PTR [esp+32], OFFSET FLAT:.LC1 ; "a==b"
        add     esp, 20
        pop     ebx
        pop     esi
        jmp     puts

```

Тем не менее, здесь 2 инструкции CMP вместо трех.

Так что, алгоритмы оптимизации GCC 4.8.1, наверное, ещё пока не идеальны.

## ARM

### 32-битный ARM

#### Оптимизирующий Keil 6/2013 (Режим ARM)

Листинг 1.115: Оптимизирующий Keil 6/2013 (Режим ARM)

```

.text:000000B8          EXPORT f_signed
.text:000000B8          f_signed          ; CODE XREF: main+C
.text:000000B8 70 40 2D E9          STMFD   SP!, {R4-R6,LR}
.text:000000BC 01 40 A0 E1          MOV     R4, R1
.text:000000C0 04 00 50 E1          CMP     R0, R4
.text:000000C4 00 50 A0 E1          MOV     R5, R0
.text:000000C8 1A 0E 8F C2          ADRGT  R0, aAB          ; "a>b\n"
.text:000000CC A1 18 00 CB          BLGT   __2printf
.text:000000D0 04 00 55 E1          CMP     R5, R4
.text:000000D4 67 0F 8F 02          ADREQ  R0, aAB_0        ; "a==b\n"
.text:000000D8 9E 18 00 0B          BLEQ   __2printf
.text:000000DC 04 00 55 E1          CMP     R5, R4
.text:000000E0 70 80 BD A8          LDMGEFD SP!, {R4-R6,PC}
.text:000000E4 70 40 BD E8          LDMFD  SP!, {R4-R6,LR}
.text:000000E8 19 0E 8F E2          ADR     R0, aAB_1        ; "a<b\n"
.text:000000EC 99 18 00 EA          B       __2printf
.text:000000EC          ; End of function f_signed

```

Многие инструкции в режиме ARM могут быть исполнены только при некоторых выставленных флагах.

Это нередко используется для сравнения чисел.

К примеру, инструкция ADD на самом деле называется ADDAL внутри, AL означает *Always*, то есть, исполнять всегда. Предикаты кодируются в 4-х старших битах инструкции 32-битных ARM-инструкций (*condition field*). Инструкция безусловного перехода B на самом деле условная и кодируется так же, как и прочие инструкции условных переходов, но имеет AL в *condition field*, то есть исполняется всегда (*execute ALways*), игнорируя флаги.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Инструкция `ADRGT` работает так же, как и `ADR`, но исполняется только в случае, если предыдущая инструкция `CMP`, сравнивая два числа, обнаруживает, что одно из них больше второго (*Greater Than*).

Следующая инструкция `BLGT` ведет себя так же, как и `BL` и сработает, только если результат сравнения “больше чем” (*Greater Than*). `ADRGT` записывает в `R0` указатель на строку `a>b\n`, а `BLGT` вызывает `printf()`. Следовательно, эти инструкции с суффиксом `-GT` исполнятся только в том случае, если значение в `R0` (там `a`) было больше, чем значение в `R4` (там `b`).

Далее мы увидим инструкции `ADREQ` и `BLEQ`. Они работают так же, как и `ADR` и `BL`, но исполняются только если значения при последнем сравнении были равны. Перед ними расположен ещё один `CMP`, потому что вызов `printf()` мог испортить состояние флагов.

Далее мы увидим `LDMGEFD`. Эта инструкция работает так же, как и `LDMFD`<sup>91</sup>, но сработает только если в результате сравнения одно из значений было больше или равно второму (*Greater or Equal*). Смысл инструкции `LDMGEFD SP!, {R4-R6,PC}` в том, что это как бы эпилог функции, но он сработает только если  $a \geq b$ , только тогда работа функции закончится.

Но если это не так, то есть  $a < b$ , то исполнение дойдет до следующей инструкции `LDMFD SP!, {R4-R6,LR}`. Это ещё один эпилог функции. Эта инструкция восстанавливает состояние регистров `R4-R6`, но и **LR!** вместо **PC!**, таким образом, пока что, не делая возврата из функции.

Последние две инструкции вызывают `printf()` со строкой «`a<b\n`» в качестве единственного аргумента. Безусловный переход на `printf()` вместо возврата из функции мы уже рассматривали в секции «`printf()` с несколькими аргументами» (?? (стр. ??)).

Функция `f_unsigned` точно такая же, но там используются инструкции `ADRHI`, `BLHI`, и `LDMCSFD`. Эти предикаты (*HI = Unsigned higher, CS = Carry Set (greater than or equal)*) аналогичны рассмотренным, но служат для работы с беззнаковыми значениями.

В функции `main()` ничего нового для нас нет:

Листинг 1.116: `main()`

```
.text:00000128          EXPORT main
.text:00000128          main
.text:00000128 10 40 2D E9          STMFDP   SP!, {R4,LR}
.text:0000012C 02 10 A0 E3          MOV      R1, #2
.text:00000130 01 00 A0 E3          MOV      R0, #1
.text:00000134 DF FF FF EB          BL       f_signed
.text:00000138 02 10 A0 E3          MOV      R1, #2
.text:0000013C 01 00 A0 E3          MOV      R0, #1
.text:00000140 EA FF FF EB          BL       f_unsigned
.text:00000144 00 00 A0 E3          MOV      R0, #0
.text:00000148 10 80 BD E8          LDMFDP   SP!, {R4,PC}
.text:00000148          ; End of function main
```

<sup>91</sup>**LDMFD!**

Так, в режиме ARM можно обойтись без условных переходов.

Почему это хорошо? Читайте здесь: ?? (стр. ??).

В x86 нет аналогичной возможности, если не считать инструкцию CMOVcc, это то же что и MOV, но она срабатывает только при определенных выставленных флагах, обычно выставленных при помощи CMP во время сравнения.

### Оптимизирующий Keil 6/2013 (Режим Thumb)

Листинг 1.117: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
.text:00000072          f_signed ; CODE XREF: main+6
.text:00000072 70 B5          PUSH    {R4-R6,LR}
.text:00000074 0C 00          MOVVS   R4, R1
.text:00000076 05 00          MOVVS   R5, R0
.text:00000078 A0 42          CMP     R0, R4
.text:0000007A 02 DD          BLE     loc_82
.text:0000007C A4 A0          ADR     R0, aAB          ; "a>b\n"
.text:0000007E 06 F0 B7 F8    BL      __2printf
.text:00000082
.text:00000082          loc_82 ; CODE XREF: f_signed+8
.text:00000082 A5 42          CMP     R5, R4
.text:00000084 02 D1          BNE     loc_8C
.text:00000086 A4 A0          ADR     R0, aAB_0        ; "a==b\n"
.text:00000088 06 F0 B2 F8    BL      __2printf
.text:0000008C
.text:0000008C          loc_8C ; CODE XREF: f_signed+12
.text:0000008C A5 42          CMP     R5, R4
.text:0000008E 02 DA          BGE     locret_96
.text:00000090 A3 A0          ADR     R0, aAB_1        ; "a<b\n"
.text:00000092 06 F0 AD F8    BL      __2printf
.text:00000096
.text:00000096          locret_96 ; CODE XREF: f_signed+1C
.text:00000096 70 BD          POP     {R4-R6,PC}
.text:00000096          ; End of function f_signed
```

В режиме Thumb только инструкции B могут быть дополнены условием исполнения (*condition code*), так что код для режима Thumb выглядит привычнее.

BLE это обычный переход с условием *Less than or Equal*, BNE — *Not Equal*, BGE — *Greater than or Equal*.

Функция `f_unsigned` точно такая же, но для работы с беззнаковыми величинами там используются инструкции BLS (*Unsigned lower or same*) и BCS (*Carry Set (Greater than or equal)*).

### ARM64: Оптимизирующий GCC (Linaro) 4.9

Листинг 1.118: `f_signed()`

```
f_signed:
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; w0=a, w1=b
cmp    w0, w1
bgt    .L19    ; Branch if Greater Than (переход, если больше чем)
(a>b)
beq    .L20    ; Branch if Equal (переход, если равно) (a==b)
bge    .L15    ; Branch if Greater than or Equal (переход, если
               больше или равно) (a>=b) (здесь это невозможно)
; a<b
adrp   x0, .LC11    ; "a<b"
add    x0, x0, :lo12:.LC11
b      puts
.L19:
adrp   x0, .LC9     ; "a>b"
add    x0, x0, :lo12:.LC9
b      puts
.L15: ; попасть сюда невозможно
ret
.L20:
adrp   x0, .LC10    ; "a==b"
add    x0, x0, :lo12:.LC10
b      puts

```

Листинг 1.119: f\_unsigned()

```

f_unsigned:
stp    x29, x30, [sp, -48]!
; w0=a, w1=b
cmp    w0, w1
add    x29, sp, 0
str    x19, [sp,16]
mov    w19, w0
bhi    .L25    ; Branch if HIgher (переход, если выше) (a>b)
cmp    w19, w1
beq    .L26    ; Branch if Equal (переход, если равно) (a==b)
.L23:
bcc    .L27    ; Branch if Carry Clear (если нет переноса)(если
               меньше, чем) (a<b)
; эпилог функции, сюда попасть невозможно
ldr    x19, [sp,16]
ldp    x29, x30, [sp], 48
ret
.L27:
ldr    x19, [sp,16]
adrp   x0, .LC11    ; "a<b"
ldp    x29, x30, [sp], 48
add    x0, x0, :lo12:.LC11
b      puts
.L25:
adrp   x0, .LC9     ; "a>b"
str    x1, [x29,40]
add    x0, x0, :lo12:.LC9
bl     puts
ldr    x1, [x29,40]
cmp    w19, w1

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

    bne    .L23    ; Branch if Not Equal (переход, если не равно)
.L26:
    ldr    x19, [sp,16]
    adrp   x0, .LC10    ; "a==b"
    ldp   x29, x30, [sp], 48
    add   x0, x0, :lo12:LC10
    b     puts

```

Комментарии добавлены автором этой книги. В глаза бросается то, что компилятор не в курсе, что некоторые ситуации невозможны, поэтому кое-где в функциях остается код, который никогда не исполнится.

## Упражнение

Попробуйте вручную оптимизировать функции по размеру, убрав избыточные инструкции и не добавляя новых.

## MIPS

Одна отличительная особенность MIPS это отсутствие регистра флагов. Очевидно, так было сделано для упрощения анализа зависимости данных (data dependency).

Так что здесь есть инструкция, похожая на SETcc в x86: SLT («Set on Less Than» — установить если меньше чем, знаковая версия) и SLTU (беззнаковая версия). Эта инструкция устанавливает регистр-получатель в 1 если условие верно или в 0 в противном случае.

Затем регистр-получатель проверяется, используя инструкцию BEQ («Branch on Equal» — переход если равно) или BNE («Branch on Not Equal» — переход если не равно) и может произойти переход. Так что эта пара инструкций должна использоваться в MIPS для сравнения и перехода. Начнем со знаковой версии нашей функции:

Листинг 1.120: Неоптимизирующий GCC 4.4.5 (IDA)

```

.text:00000000 f_signed: # CODE XREF: main+18
.text:00000000
.text:00000000 var_10 = -0x10
.text:00000000 var_8 = -8
.text:00000000 var_4 = -4
.text:00000000 arg_0 = 0
.text:00000000 arg_4 = 4
.text:00000000
.text:00000000      addiu   $sp, -0x20
.text:00000004      sw     $ra, 0x20+var_4($sp)
.text:00000008      sw     $fp, 0x20+var_8($sp)
.text:0000000C      move   $fp, $sp
.text:00000010      la    $gp, __gnu_local_gp
.text:00000018      sw     $gp, 0x20+var_10($sp)
; сохранить входные значения в локальном стеке:
.text:0000001C      sw     $a0, 0x20+arg_0($fp)

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:00000020      sw      $a1, 0x20+arg_4($fp)
; перезагрузить их:
.text:00000024      lw      $v1, 0x20+arg_0($fp)
.text:00000028      lw      $v0, 0x20+arg_4($fp)
; $v0=b
; $v1=a
.text:0000002C      or      $at, $zero ; NOP
; это псевдоинструкция. на самом деле, там "slt $v0,$v0,$v1" .
; так что $v0 будет установлен в 1, если $v0<$v1 (b<a) или в 0 в противном
; случае:
.text:00000030      slt     $v0, $v1
; перейти на loc_5c, если условие не верно.
; это псевдоинструкция. на самом деле, там "beq $v0,$zero,loc_5c" :
.text:00000034      beqz   $v0, loc_5C
; вывести "a>b" и выйти
.text:00000038      or     $at, $zero ; branch delay slot, NOP
.text:0000003C      lui    $v0, (unk_230 >> 16) # "a>b"
.text:00000040      addiu  $a0, $v0, (unk_230 & 0xFFFF) # "a>b"
.text:00000044      lw     $v0, (puts & 0xFFFF)($gp)
.text:00000048      or     $at, $zero ; NOP
.text:0000004C      move  $t9, $v0
.text:00000050      jalr  $t9
.text:00000054      or     $at, $zero ; branch delay slot, NOP
.text:00000058      lw     $gp, 0x20+var_10($fp)
.text:0000005C      loc_5C:                                # CODE XREF: f_signed+34
.text:0000005C      lw     $v1, 0x20+arg_0($fp)
.text:00000060      lw     $v0, 0x20+arg_4($fp)
.text:00000064      or     $at, $zero ; NOP
; проверить a==b, перейти на loc_90, если это не так:
.text:00000068      bne   $v1, $v0, loc_90
.text:0000006C      or     $at, $zero ; branch delay slot, NOP
; условие верно, вывести "a==b" и закончить:
.text:00000070      lui    $v0, (aAB >> 16) # "a==b"
.text:00000074      addiu  $a0, $v0, (aAB & 0xFFFF) # "a==b"
.text:00000078      lw     $v0, (puts & 0xFFFF)($gp)
.text:0000007C      or     $at, $zero ; NOP
.text:00000080      move  $t9, $v0
.text:00000084      jalr  $t9
.text:00000088      or     $at, $zero ; branch delay slot, NOP
.text:0000008C      lw     $gp, 0x20+var_10($fp)
.text:00000090      loc_90:                                # CODE XREF: f_signed+68
.text:00000090      lw     $v1, 0x20+arg_0($fp)
.text:00000094      lw     $v0, 0x20+arg_4($fp)
.text:00000098      or     $at, $zero ; NOP
; проверить условие $v1<$v0 (a<b), установить $v0 в 1, если условие верно:
.text:0000009C      slt   $v0, $v1, $v0
; если условие не верно (т.е. $v0==0), перейти на loc_c8:
.text:000000A0      beqz  $v0, loc_C8
.text:000000A4      or     $at, $zero ; branch delay slot, NOP
; условие верно, вывести "a<b" и закончить
.text:000000A8      lui    $v0, (aAB_0 >> 16) # "a<b"
.text:000000AC      addiu  $a0, $v0, (aAB_0 & 0xFFFF) # "a<b"

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:000000B0      lw      $v0, (puts & 0xFFFF)($gp)
.text:000000B4      or      $at, $zero ; NOP
.text:000000B8      move   $t9, $v0
.text:000000BC      jalr   $t9
.text:000000C0      or      $at, $zero ; branch delay slot, NOP
.text:000000C4      lw      $gp, 0x20+var_10($fp)
.text:000000C8
; все 3 условия были неверны, так что просто заканчиваем:
.text:000000C8 loc_C8:                                     # CODE XREF:
                f_signed+A0
.text:000000C8      move   $sp, $fp
.text:000000CC      lw      $ra, 0x20+var_4($sp)
.text:000000D0      lw      $fp, 0x20+var_8($sp)
.text:000000D4      addiu  $sp, 0x20
.text:000000D8      jr     $ra
.text:000000DC      or      $at, $zero ; branch delay slot, NOP
.text:000000DC      # End of function f_signed

```

SLT REG0, REG0, REG1 сокращается в IDA до более короткой формы SLT REG0, REG1. Мы также видим здесь псевдоинструкцию BEQZ («Branch if Equal to Zero» — переход если равно нулю), которая, на самом деле, BEQ REG, \$ZERO, LABEL.

Беззнаковая версия точно такая же, только здесь используется SLTU (беззнаковая версия, отсюда «U» в названии) вместо SLT:

Листинг 1.121: Неоптимизирующий GCC 4.4.5 (IDA)

```

.text:000000E0 f_unsigned: # CODE XREF: main+28
.text:000000E0
.text:000000E0 var_10 = -0x10
.text:000000E0 var_8 = -8
.text:000000E0 var_4 = -4
.text:000000E0 arg_0 = 0
.text:000000E0 arg_4 = 4
.text:000000E0
.text:000000E0      addiu  $sp, -0x20
.text:000000E4      sw     $ra, 0x20+var_4($sp)
.text:000000E8      sw     $fp, 0x20+var_8($sp)
.text:000000EC      move  $fp, $sp
.text:000000F0      la    $gp, __gnu_local_gp
.text:000000F8      sw     $gp, 0x20+var_10($sp)
.text:000000FC      sw     $a0, 0x20+arg_0($fp)
.text:00000100      sw     $a1, 0x20+arg_4($fp)
.text:00000104      lw     $v1, 0x20+arg_0($fp)
.text:00000108      lw     $v0, 0x20+arg_4($fp)
.text:0000010C      or     $at, $zero
.text:00000110      sltu  $v0, $v1
.text:00000114      beqz  $v0, loc_13C
.text:00000118      or     $at, $zero
.text:0000011C      lui   $v0, (unk_230 >> 16)
.text:00000120      addiu $a0, $v0, (unk_230 & 0xFFFF)
.text:00000124      lw     $v0, (puts & 0xFFFF)($gp)
.text:00000128      or     $at, $zero
.text:0000012C      move  $t9, $v0

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:00000130      jalr    $t9
.text:00000134      or     $at, $zero
.text:00000138      lw     $gp, 0x20+var_10($fp)
.text:0000013C
.text:0000013C loc_13C:          # CODE XREF: f_unsigned+34
.text:0000013C      lw     $v1, 0x20+arg_0($fp)
.text:00000140      lw     $v0, 0x20+arg_4($fp)
.text:00000144      or     $at, $zero
.text:00000148      bne   $v1, $v0, loc_170
.text:0000014C      or     $at, $zero
.text:00000150      lui   $v0, (aAB >> 16) # "a==b"
.text:00000154      addiu $a0, $v0, (aAB & 0xFFFF) # "a==b"
.text:00000158      lw     $v0, (puts & 0xFFFF)($gp)
.text:0000015C      or     $at, $zero
.text:00000160      move  $t9, $v0
.text:00000164      jalr  $t9
.text:00000168      or     $at, $zero
.text:0000016C      lw     $gp, 0x20+var_10($fp)
.text:00000170
.text:00000170 loc_170:          # CODE XREF: f_unsigned+68
.text:00000170      lw     $v1, 0x20+arg_0($fp)
.text:00000174      lw     $v0, 0x20+arg_4($fp)
.text:00000178      or     $at, $zero
.text:0000017C      sltu  $v0, $v1, $v0
.text:00000180      beqz  $v0, loc_1A8
.text:00000184      or     $at, $zero
.text:00000188      lui   $v0, (aAB_0 >> 16) # "a<b"
.text:0000018C      addiu $a0, $v0, (aAB_0 & 0xFFFF) # "a<b"
.text:00000190      lw     $v0, (puts & 0xFFFF)($gp)
.text:00000194      or     $at, $zero
.text:00000198      move  $t9, $v0
.text:0000019C      jalr  $t9
.text:000001A0      or     $at, $zero
.text:000001A4      lw     $gp, 0x20+var_10($fp)
.text:000001A8
.text:000001A8 loc_1A8:          # CODE XREF: f_unsigned+A0
.text:000001A8      move  $sp, $fp
.text:000001AC      lw     $ra, 0x20+var_4($sp)
.text:000001B0      lw     $fp, 0x20+var_8($sp)
.text:000001B4      addiu $sp, 0x20
.text:000001B8      jr    $ra
.text:000001BC      or     $at, $zero
.text:000001BC      # End of function f_unsigned

```

### 1.18.2. Вычисление абсолютной величины

Это простая функция:

```

int my_abs (int i)
{
    if (i<0)
        return -i;
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        else
            return i;
};

```

### Оптимизирующий MSVC

Обычный способ генерации кода:

Листинг 1.122: Оптимизирующий MSVC 2012 x64

```

i$ = 8
my_abs PROC
; ECX = input
    test    ecx, ecx
; проверить знак входного значения
; пропустить инструкцию NEG, если знак положительный
    jns    SHORT $LN2@my_abs
; поменять знак
    neg    ecx
$LN2@my_abs:
; подготовить результат в EAX:
    mov    eax, ecx
    ret    0
my_abs ENDP

```

GCC 4.9 делает почти то же самое.

### Оптимизирующий Keil 6/2013: Режим Thumb

Листинг 1.123: Оптимизирующий Keil 6/2013: Режим Thumb

```

my_abs PROC
    CMP    r0, #0
; входное значение равно нулю или больше нуля?
; в таком случае, пропустить инструкцию RSBS
    BGE    |L0.6|
; отнять входное значение от 0:
    RSBS   r0, r0, #0
|L0.6|
    BX    lr
    ENDP

```

В ARM нет инструкции для изменения знака, так что компилятор Keil использует инструкцию «Reverse Subtract», которая просто вычитает, но с операндами, переставленными наоборот.

### Оптимизирующий Keil 6/2013: Режим ARM

В режиме ARM можно добавлять коды условий к некоторым инструкциям, что компилятор Keil и сделал:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

## Листинг 1.124: Оптимизирующий Keil 6/2013: Режим ARM

```

my_abs PROC
    CMP        r0,#0
; исполнить инструкцию "Reverse Subtract" только в случае,
; если входное значение меньше 0:
    RSBLT     r0,r0,#0
    BX        lr
ENDP

```

Теперь здесь нет условных переходов и это хорошо:

?? (стр. ??).

**Неоптимизирующий GCC 4.9 (ARM64)**

В ARM64 есть инструкция NEG для смены знака:

## Листинг 1.125: Оптимизирующий GCC 4.9 (ARM64)

```

my_abs:
    sub      sp, sp, #16
    str      w0, [sp,12]
    ldr      w0, [sp,12]
; сравнить входное значение с содержимым регистра WZR
; (который всегда содержит ноль)
    cmp      w0, wzr
    bge      .L2
    ldr      w0, [sp,12]
    neg      w0, w0
    b        .L3
.L2:
    ldr      w0, [sp,12]
.L3:
    add      sp, sp, 16
    ret

```

**MIPS**

## Листинг 1.126: Оптимизирующий GCC 4.4.5 (IDA)

```

my_abs:
; перейти если $a0<0:
    bltz    $a0, locret_10
; просто вернуть входное значение ($a0) в $v0:
    move    $v0, $a0
    jr      $ra
    or      $at, $zero ; branch delay slot, NOP
locret_10:
; поменять у значения знак и сохранить его в $v0:
    jr      $ra
; это псевдоинструкция. на самом деле, это "subu $v0,$zero,$a0" ($v0=0-$a0)
    negu    $v0, $a0

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Видим здесь новую инструкцию: BLTZ («Branch if Less Than Zero»). Тут есть также псевдоинструкция NEGU, которая на самом деле вычитает из нуля. Суффикс «U» в обеих инструкциях SUBU и NEGU означает, что при целочисленном переполнении исключение не работает.

### Версия без переходов?

Возможна также версия и без переходов, мы рассмотрим её позже: ?? (стр. ??).

### 1.18.3. Тернарный условный оператор

Тернарный условный оператор (ternary conditional operator) в Си/Си++ это:

```
expression ? expression : expression
```

И вот пример:

```
const char* f (int a)
{
    return a==10 ? "it is ten" : "it is not ten";
};
```

### x86

Старые и неоптимизирующие компиляторы генерируют код так, как если бы выражение if/else было использовано вместо него:

Листинг 1.127: Неоптимизирующий MSVC 2008

```
$SG746 DB      'it is ten', 00H
$SG747 DB      'it is not ten', 00H

tv65 = -4 ; будет использовано как временная переменная
_a$ = 8
_f      PROC
        push    ebp
        mov     ebp, esp
        push    ecx
; сравнить входное значение с 10
        cmp     DWORD PTR _a$[ebp], 10
; переход на $LN3@f если не равно
        jne     SHORT $LN3@f
; сохранить указатель на строку во временной переменной:
        mov     DWORD PTR tv65[ebp], OFFSET $SG746 ; 'it is ten'
; перейти на выход
        jmp     SHORT $LN4@f
$LN3@f:
; сохранить указатель на строку во временной переменной:
        mov     DWORD PTR tv65[ebp], OFFSET $SG747 ; 'it is not ten'
$LN4@f:
; это выход. скопировать указатель на строку из временной переменной в EAX.
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        mov     eax, DWORD PTR tv65[ebp]
        mov     esp, ebp
        pop     ebp
        ret     0
_f      ENDP

```

Листинг 1.128: Оптимизирующий MSVC 2008

```

$SG792 DB      'it is ten', 00H
$SG793 DB      'it is not ten', 00H

_a$ = 8 ; size = 4
_f      PROC
; сравнить входное значение с 10
        cmp     DWORD PTR _a$[esp-4], 10
        mov     eax, OFFSET $SG792 ; 'it is ten'
; переход на $LN4@f если равно
        je     SHORT $LN4@f
        mov     eax, OFFSET $SG793 ; 'it is not ten'
$LN4@f:
        ret     0
_f      ENDP

```

Новые компиляторы могут быть более краткими:

Листинг 1.129: Оптимизирующий MSVC 2012 x64

```

$SG1355 DB     'it is ten', 00H
$SG1356 DB     'it is not ten', 00H

a$ = 8
f      PROC
; загрузить указатели на обе строки
        lea     rdx, OFFSET FLAT:$SG1355 ; 'it is ten'
        lea     rax, OFFSET FLAT:$SG1356 ; 'it is not ten'
; сравнить входное значение с 10
        cmp     ecx, 10
; если равно, скопировать значение из RDX ("it is ten")
; если нет, ничего не делаем. указатель на строку
; "it is not ten" всё еще в RAX.
        cmov   rax, rdx
        ret     0
f      ENDP

```

Оптимизирующий GCC 4.8 для x86 также использует инструкцию `CMOVcc`, тогда как неоптимизирующий GCC 4.8 использует условные переходы.

## ARM

Оптимизирующий Keil для режима ARM также использует инструкцию `ADRcc`, срабатывающую при некотором условии:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



Листинг 1.130: Оптимизирующий Keil 6/2013 (Режим ARM)

```
f PROC
; сравнить входное значение с 10
    CMP    r0,#0xa
; если результат сравнения EQual (равно),
; скопировать указатель на строку "it is ten" в R0
    ADREQ  r0,|L0.16| ; "it is ten"
; если результат сравнения Not Equal (не равно),
; скопировать указатель на строку "it is not ten" в R0
    ADRNE  r0,|L0.28| ; "it is not ten"
    BX     lr
    ENDP

|L0.16|
    DCB    "it is ten",0
|L0.28|
    DCB    "it is not ten",0
```

Без внешнего вмешательства инструкции ADREQ и ADRNE никогда не исполнятся одновременно. Оптимизирующий Keil для режима Thumb вынужден использовать инструкции условного перехода, потому что тут нет инструкции загрузки значения, поддерживающей флаги условия:

Листинг 1.131: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
f PROC
; сравнить входное значение с 10
    CMP    r0,#0xa
; переход на |L0.8| если EQual (равно)
    BEQ    |L0.8|
    ADR    r0,|L0.12| ; "it is not ten"
    BX     lr
|L0.8|
    ADR    r0,|L0.28| ; "it is ten"
    BX     lr
    ENDP

|L0.12|
    DCB    "it is not ten",0
|L0.28|
    DCB    "it is ten",0
```

## ARM64

Оптимизирующий GCC (Linaro) 4.9 для ARM64 также использует условные переходы:

Листинг 1.132: Оптимизирующий GCC (Linaro) 4.9

```
f:
    cmp    x0, 10
    beq    .L3          ; branch if equal (переход, если равно)
    adrp   x0, .LC1     ; "it is ten"
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        add    x0, x0, :lo12:.LC1
        ret
.L3:
        adrp   x0, .LC0          ; "it is not ten"
        add    x0, x0, :lo12:.LC0
        ret
.LC0:
        .string "it is ten"
.LC1:
        .string "it is not ten"

```

Это потому что в ARM64 нет простой инструкции загрузки с флагами условия, как ADRcc в 32-битном режиме ARM или MOVcc в x86.

Но с другой стороны, там есть инструкция CSEL («Conditional SElect») [ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, (2013)p390, C5.5], но GCC 4.9 наверное, пока не так хорош, чтобы генерировать её в таком фрагменте кода

## MIPS

GCC 4.4.5 для MIPS тоже не так хорош, к сожалению:

Листинг 1.133: Оптимизирующий GCC 4.4.5 (вывод на ассемблере)

```

$LC0:
        .ascii "it is not ten\000"
$LC1:
        .ascii "it is ten\000"
f:
        li     $2,10                # 0xa
; сравнить $a0 и 10, переход, если равно:
        beq   $4,$2,$L2
        nop   ; branch delay slot

; оставить адрес строки "it is not ten" в $v0 и выйти:
        lui   $2,%hi($LC0)
        j     $31
        addiu $2,$2,%lo($LC0)

$L2:
; оставить адрес строки "it is ten" в $v0 и выйти:
        lui   $2,%hi($LC1)
        j     $31
        addiu $2,$2,%lo($LC1)

```

## Перепишем, используя обычный if/else

```

const char* f (int a)
{
    if (a==10)
        return "it is ten";
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        else
            return "it is not ten";
};

```

Интересно, оптимизирующий GCC 4.8 для x86 также может генерировать MOVSS в этом случае:

Листинг 1.134: Оптимизирующий GCC 4.8

```

.LC0:
    .string "it is ten"
.LC1:
    .string "it is not ten"
f:
.LFB0:
; сравнить входное значение с 10
    cmp    DWORD PTR [esp+4], 10
    mov    edx, OFFSET FLAT:.LC1 ; "it is not ten"
    mov    eax, OFFSET FLAT:.LC0 ; "it is ten"
; если результат сравнение Not Equal (не равно), скопировать значение из EDX
  в EAX
; а если нет, то ничего не делать
    cmovne eax, edx
    ret

```

Оптимизирующий Keil в режиме ARM генерирует код идентичный этому: листинг.??  
Но оптимизирующий MSVC 2012 пока не так хорош.

## Вывод

Почему оптимизирующие компиляторы стараются избавиться от условных переходов? Читайте больше об этом здесь: ?? (стр. ??).

## 1.18.4. Поиск минимального и максимального значения

### 32-bit

```

int my_max(int a, int b)
{
    if (a>b)
        return a;
    else
        return b;
};

int my_min(int a, int b)
{
    if (a<b)
        return a;
    else
        return b;
};

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

## Листинг 1.135: Неоптимизирующий MSVC 2013

```

_a$ = 8
_b$ = 12
_my_min PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
; сравнить A и B:
    cmp     eax, DWORD PTR _b$[ebp]
; переход, если A больше или равно B:
    jge     SHORT $LN2@my_min
; перезагрузить A в EAX в противном случае и перейти на выход
    mov     eax, DWORD PTR _a$[ebp]
    jmp     SHORT $LN3@my_min
    jmp     SHORT $LN3@my_min ; это избыточная инструкция
$LN2@my_min:
; возврат B
    mov     eax, DWORD PTR _b$[ebp]
$LN3@my_min:
    pop     ebp
    ret     0
_my_min ENDP

_a$ = 8
_b$ = 12
_my_max PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
; сравнить A и B:
    cmp     eax, DWORD PTR _b$[ebp]
; переход, если A меньше или равно B:
    jle     SHORT $LN2@my_max
; перезагрузить A в EAX в противном случае и перейти на выход
    mov     eax, DWORD PTR _a$[ebp]
    jmp     SHORT $LN3@my_max
    jmp     SHORT $LN3@my_max ; это избыточная инструкция
$LN2@my_max:
; возврат B
    mov     eax, DWORD PTR _b$[ebp]
$LN3@my_max:
    pop     ebp
    ret     0
_my_max ENDP

```

Эти две функции отличаются друг от друга только инструкцией условного перехода: JGE («Jump if Greater or Equal» — переход если больше или равно) используется в первой и JLE («Jump if Less or Equal» — переход если меньше или равно) во второй.

Здесь есть ненужная инструкция JMP в каждой функции, которую MSVC, наверное, оставил по ошибке.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

**Без переходов**

ARM в режиме Thumb напоминает нам x86-код:

Листинг 1.136: Оптимизирующий Keil 6/2013 (Режим Thumb)

```

my_max PROC
; R0=A
; R1=B
; сравнить A и B:
    CMP    r0,r1
; переход, если A больше B:
    BGT   |L0.6|
; в противном случае (A<=B) возврат R1 (B):
    MOVS  r0,r1
|L0.6|
; возврат
    BX    lr
    ENDP

my_min PROC
; R0=A
; R1=B
; сравнить A и B:
    CMP    r0,r1
; переход, если A меньше B:
    BLT   |L0.14|
; в противном случае (A>=B) возврат R1 (B):
    MOVS  r0,r1
|L0.14|
; возврат
    BX    lr
    ENDP

```

Функции отличаются только инструкцией перехода: BGT и BLT. А в режиме ARM можно использовать условные суффиксы, так что код более плотный. MOVcc будет исполнена только если условие верно:

Листинг 1.137: Оптимизирующий Keil 6/2013 (Режим ARM)

```

my_max PROC
; R0=A
; R1=B
; сравнить A и B:
    CMP    r0,r1
; вернуть B вместо A копируя B в R0
; эта инструкция сработает только если A<=B (т.е. LE - Less or Equal, меньше
или равно)
; если инструкция не сработает (в случае A>B),
; A всё ещё в регистре R0
    MOVLE  r0,r1
    BX    lr
    ENDP

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

my_min PROC
; R0=A
; R1=B
; сравнить A и B:
    CMP    r0,r1
; вернуть B вместо A копируя B в R0
; эта инструкция сработает только если A>=B (т.е. GE - Greater or Equal,
    больше или равно)
; если инструкция не сработает (в случае A<B),
; A всё ещё в регистре R0
    MOVGE  r0,r1
    BX     lr
    ENDP

```

Оптимизирующий GCC 4.8.1 и оптимизирующий MSVC 2013 могут использовать инструкцию `CMOVcc`, которая аналогична `MOVcc` в ARM:

Листинг 1.138: Оптимизирующий MSVC 2013

```

my_max:
    mov     edx, DWORD PTR [esp+4]
    mov     eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; сравнить A и B:
    cmp     edx, eax
; если A>=B, загрузить значение A в EAX
; в противном случае, эта инструкция ничего не делает (если A<B)
    cmovge eax, edx
    ret

my_min:
    mov     edx, DWORD PTR [esp+4]
    mov     eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; сравнить A и B:
    cmp     edx, eax
; если A<=B, загрузить значение A в EAX
; в противном случае, эта инструкция ничего не делает (если A>B)
    cmovle eax, edx
    ret

```

## 64-bit

```

#include <stdint.h>

int64_t my_max(int64_t a, int64_t b)
{
    if (a>b)
        return a;
    else
        return b;
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

};

int64_t my_min(int64_t a, int64_t b)
{
    if (a<b)
        return a;
    else
        return b;
};

```

Тут есть ненужные перетасовки значений, но код в целом понятен:

Листинг 1.139: Неоптимизирующий GCC 4.9.1 ARM64

```

my_max:
    sub    sp, sp, #16
    str    x0, [sp,8]
    str    x1, [sp]
    ldr    x1, [sp,8]
    ldr    x0, [sp]
    cmp    x1, x0
    ble    .L2
    ldr    x0, [sp,8]
    b      .L3
.L2:
    ldr    x0, [sp]
.L3:
    add    sp, sp, 16
    ret

my_min:
    sub    sp, sp, #16
    str    x0, [sp,8]
    str    x1, [sp]
    ldr    x1, [sp,8]
    ldr    x0, [sp]
    cmp    x1, x0
    bge    .L5
    ldr    x0, [sp,8]
    b      .L6
.L5:
    ldr    x0, [sp]
.L6:
    add    sp, sp, 16
    ret

```

## Без переходов

Нет нужды загружать аргументы функции из стека, они уже в регистрах:

Листинг 1.140: Оптимизирующий GCC 4.9.1 x64

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

my_max:
; RDI=A
; RSI=B
; сравнить A и B:
    cmp    rdi, rsi
; подготовить B в RAX для возврата:
    mov    rax, rsi
; если A>=B, оставить A (RDI) в RAX для возврата.
; в противном случае, инструкция ничего не делает (если A<B), и в RAX
    остается B
    cmovge rax, rdi
    ret

my_min:
; RDI=A
; RSI=B
; сравнить A и B:
    cmp    rdi, rsi
; подготовить B в RAX для возврата:
    mov    rax, rsi
; если A<=B, оставить A (RDI) в RAX для возврата.
; в противном случае, инструкция ничего не делает (если A>B), и в RAX
    остается B
    cmovle rax, rdi
    ret

```

MSVC 2013 делает то же самое.

В ARM64 есть инструкция CSEL, которая работает точно также, как и MOVcc в ARM и CMOVcc в x86, но название другое: «Conditional SElect».

#### Листинг 1.141: Оптимизирующий GCC 4.9.1 ARM64

```

my_max:
; X0=A
; X1=B
; сравнить A и B:
    cmp    x0, x1
; выбрать X0 (A) в X0 если X0>=X1 или A>=B (Greater or Equal: больше или
    равно)
; выбрать X1 (B) в X0 если A<B
    csel   x0, x0, x1, ge
    ret

my_min:
; X0=A
; X1=B
; сравнить A и B:
    cmp    x0, x1
; выбрать X0 (A) в X0 если X0<=X1 (Less or Equal: меньше или равно)
; выбрать X1 (B) в X0 если A>B
    csel   x0, x0, x1, le
    ret

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



**MIPS**

А GCC 4.4.5 для MIPS не так хорош, к сожалению:

Листинг 1.142: Оптимизирующий GCC 4.4.5 (IDA)

```

my_max:
; установить $v1 в 1, если $a1 < $a0, в противном случае очистить (если
; $a1 > $a0):
        slt    $v1, $a1, $a0
; переход, если в $v1 ноль (или $a1 > $a0):
        beqz   $v1, locret_10
; это branch delay slot
; подготовить $a1 в $v0 на случай, если переход сработает:
        move   $v0, $a1
; переход не сработал, подготовить $a0 в $v0:
        move   $v0, $a0

locret_10:
        jr     $ra
        or     $at, $zero ; branch delay slot, NOP

; функция min() точно такая же,
; но входные операнды в инструкции SLT поменяны местами:
my_min:
        slt    $v1, $a0, $a1
        beqz   $v1, locret_28
        move   $v0, $a1
        move   $v0, $a0

locret_28:
        jr     $ra
        or     $at, $zero ; branch delay slot, NOP

```

Не забывайте о *branch delay slots*: первая MOVE исполняется *перед* BEQZ, вторая MOVE исполняется только если переход не произошел.

**1.18.5. Вывод****x86**

Примерный скелет условных переходов:

Листинг 1.143: x86

```

CMP register, register/value
Jcc true ; cc=код условия
false:
;... код, исполняющийся, если сравнение ложно ...
JMP exit
true:
;... код, исполняющийся, если сравнение истинно ...
exit:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

**ARM**

Листинг 1.144: ARM

```
CMP register, register/value
Bcc true ; cc=код условия
false:
;... код, исполняющийся, если сравнение ложно ...
JMP exit
true:
;... код, исполняющийся, если сравнение истинно ...
exit:
```

**MIPS**

Листинг 1.145: Проверка на ноль

```
BEQZ REG, label
...
```

Листинг 1.146: Меньше ли нуля? (используя псевдоинструкцию)

```
BLTZ REG, label
...
```

Листинг 1.147: Проверка на равенство

```
BEQ REG1, REG2, label
...
```

Листинг 1.148: Проверка на неравенство

```
BNE REG1, REG2, label
...
```

Листинг 1.149: Проверка на меньше (знаковое)

```
SLT REG1, REG2, REG3
BEQ REG1, label
...
```

Листинг 1.150: Проверка на меньше (беззнаковое)

```
SLTU REG1, REG2, REG3
BEQ REG1, label
...
```

**Без инструкций перехода**

Если тело условного выражения очень короткое, может быть использована инструкция условного копирования: MOVcc в ARM (в режиме ARM), CSEL в ARM64, CMOVcc в x86.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

**ARM**

В режиме ARM можно использовать условные суффиксы для некоторых инструкций:

Листинг 1.151: ARM (Режим ARM)

```
CMP register, register/value
instr1_cc ; инструкция, которая будет исполнена, если условие истинно
instr2_cc ; еще инструкция, которая будет исполнена, если условие истинно
;... и т.д....
```

Нет никаких ограничений на количество инструкций с условными суффиксами до тех пор, пока флаги CPU не были модифицированы одной из таких инструкций.

В режиме Thumb есть инструкция IT, позволяющая дополнить следующие 4 инструкции суффиксами, задающими условие.

Читайте больше об этом: ?? (стр. ??).

Листинг 1.152: ARM (Режим Thumb)

```
CMP register, register/value
ITEEE EQ ; выставить такие суффиксы: if-then-else-else-else
instr1 ; инструкция будет исполнена, если истинно
instr2 ; инструкция будет исполнена, если ложно
instr3 ; инструкция будет исполнена, если ложно
instr4 ; инструкция будет исполнена, если ложно
```

**1.18.6. Упражнение**

(ARM64) Попробуйте переписать код в листинг ?? убрав все инструкции условного перехода, и используйте инструкцию CSEL.

**1.19. Взлом ПО**

Огромная часть ПО взламывается таким образом — поиском того самого места, где проверяется защита, донгла (?? (стр. ??)), лицензионный ключ, серийный номер, и т. д.

Очень часто это так и выглядит:

```
...
call check_protection
jz all_OK
call message_box_protection_missing
call exit
all_OK:
; proceed
...
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Так что если вы видите патч (или “крэк”), взламывающий некое ПО, и этот патч заменяет байт(ы) 0x74/0x75 (JZ/JNZ) на 0xEB (JMP), то это оно и есть.

Собственно, процесс взлома большинства ПО сводится к поиску того самого JMP-а.

Но бывает и так, что ПО проверяет защиту время от времени, это может быть донгла, или ПО может через интернет проверять сервер лицензий. Тогда приходится искать ф-цию, проверяющую защиту. Затем пропатчить, вписав туда `xor eax, eax / retn`, либо `mov eax, 1 / retn`.

Важно понимать, что пропатчив начало ф-ции двумя инструкциями, обычно, за ними остается некий мусор. Он состоит из куска одной из инструкций и нескольких последующих инструкций.

Вот реальный пример. Начало некоей ф-ции, которую мы хотим *заменить* на `return 1;`

Листинг 1.153: Было

8BFF	mov	edi,edi
55	push	ebp
8BEC	mov	ebp,esp
81EC68080000	sub	esp,000000868
A110C00001	mov	eax,[00100C010]
33C5	xor	eax,ebp
8945FC	mov	[ebp][-4],eax
53	push	ebx
8B5D08	mov	ebx,[ebp][8]
...		

Листинг 1.154: Стало

B801000000	mov	eax,1
C3	retn	
EC	in	al,dx
68080000A1	push	0A1000008
10C0	adc	al,al
0001	add	[ecx],al
33C5	xor	eax,ebp
8945FC	mov	[ebp][-4],eax
53	push	ebx
8B5D08	mov	ebx,[ebp][8]
...		

Появляются несколько некорректных инструкций — IN, PUSH, ADC, ADD, затем дизассемблер Niew (котором я только что воспользовался) синхронизируется и продолжает дизассемблировать корректно остальную часть ф-ции.

Всё это не важно — все эти инструкции следующие за RETN не будут исполняться никогда, если только на какую-то из них не будет прямого перехода откуда-то, чего, конечно, в общем случае, не будет.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Также, нередко бывает некая глобальная булева переменная, хранящая флаг, зарегистрированно ли ПО или нет.

```
init_etc proc
...
call check_protection_or_license_file
mov is_demo, eax
...
retn
init_etc endp

...

save_file proc
...
mov eax, is_demo
cmp eax, 1
jz all_OK1

call message_box_it_is_a_demo_no_saving_allowed
retn

:all_OK1
; continue saving file

...

save_proc endp

somewhere_else proc

mov eax, is_demo
cmp eax, 1
jz all_OK

; check if we run for 15 minutes
; exit if it is so
; or show nagging screen

:all_OK2
; continue

somewhere_else endp
```

Тут можно или патчить начало ф-ции `check_protection_or_license_file()`, чтобы она всегда возвращала 1, либо, если вдруг так лучше, патчить все инструкции `JZ/JNZ`.

Еще немного о модификации файлов: ??.

## 1.20. Пранк: невозможность выйти из Windows 7

Я уже не помню, как я нашел ф-цию `ExitWindowsEx()` в Windows 98 (это был конец 1990-х), в файле `user32.dll`. Вероятно я просто заметил само себя описывающее имя. И затем я попробовал *заблокировать* ей изменив первый байт на `0xC3` (`RETN`).

В итоге стало смешно: из Windows 98 нельзя было выйти. Пришлось нажимать на кнопку `reset`.

И вот теперь, на днях, я попробовал сделать то же самое в Windows 7, созданную почти на 10 лет позже, на базе принципиально другой Windows NT. И все еще, ф-ция `ExitWindowsEx()` присутствует в файле `user32.dll` и служит тем же целям.

В начале, я отключил *Windows File Protection* добавив это в реестр (а иначе Windows будет молча восстанавливать модифицированные системные файлы):

```
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon]
"SFCDisable"=dword:ffffff9d
```

Затем я переименовал `c:\windows\system32\user32.dll` в `user32.dll.bak`. Я нашел точку входа ф-ции `ExitWindowsEx()` в списке экспортируемых адресов в `Hiew` (то же можно было бы сделать и в `IDA`) и записал туда байт `0xC3`. Я перезагрузил Windows 7 и теперь её нельзя *зашатдаунить*. Кнопки "Restart" и "Logoff" больше не работают.

Не знаю, смешно ли это в наше время или нет, но тогда, в конце 90-х, мой товарищ отнес пропатченный файл `user32.dll` на дискете в свой университет и скопировал его на все компьютеры (бывшие в его доступе, работавшие под Windows 98 (почти все)). После этого, выйти корректно из Windows было нельзя и его преподаватель информатики был адово злой. (Надеюсь, он мог бы простить нас, если он это сейчас читает.)

Если вы делаете это, сохраните оригиналы всех файлов. Лучше всего, запускать Windows в виртуальной машине.

## 1.21. `switch()/case/default`

### 1.21.1. Если вариантов мало

```
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
        case 0: printf ("zero\n"); break;
        case 1: printf ("one\n"); break;
        case 2: printf ("two\n"); break;
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        default: printf ("something unknown\n"); break;
    };
};

int main()
{
    f (2); // test
};

```

## x86

### Неоптимизирующий MSVC

Это дает в итоге (MSVC 2010):

Листинг 1.155: MSVC 2010

```

tv64 = -4 ; size = 4
_a$ = 8   ; size = 4
_f      PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _a$[ebp]
    mov     DWORD PTR tv64[ebp], eax
    cmp     DWORD PTR tv64[ebp], 0
    je     SHORT $LN4@f
    cmp     DWORD PTR tv64[ebp], 1
    je     SHORT $LN3@f
    cmp     DWORD PTR tv64[ebp], 2
    je     SHORT $LN2@f
    jmp     SHORT $LN1@f
$LN4@f:
    push    OFFSET $SG739 ; 'zero', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN7@f
$LN3@f:
    push    OFFSET $SG741 ; 'one', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN7@f
$LN2@f:
    push    OFFSET $SG743 ; 'two', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN7@f
$LN1@f:
    push    OFFSET $SG745 ; 'something unknown', 0aH, 00H
    call    _printf
    add     esp, 4
$LN7@f:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov    esp, ebp
pop    ebp
ret    0
_f     ENDP

```

Наша функция с оператором `switch()`, с небольшим количеством вариантов, это практически аналог подобной конструкции:

```

void f (int a)
{
    if (a==0)
        printf ("zero\n");
    else if (a==1)
        printf ("one\n");
    else if (a==2)
        printf ("two\n");
    else
        printf ("something unknown\n");
};

```

Когда вариантов немного и мы видим подобный код, невозможно сказать с уверенностью, был ли в оригинальном исходном коде `switch()`, либо просто набор операторов `if()`.

То есть, `switch()` это синтаксический сахар для большого количества вложенных проверок при помощи `if()`.

В самом выходном коде ничего особо нового, за исключением того, что компилятор зачем-то перекладывает входящую переменную (*a*) во временную в локальном стеке `v64`<sup>92</sup>.

Если скомпилировать это при помощи GCC 4.4.1, то будет почти то же самое, даже с максимальной оптимизацией (ключ `-O3`).

### Оптимизирующий MSVC

Попробуем включить оптимизацию кодегенератора MSVC (`/Ox`): `cl 1.c /Fa1.asm /Ox`

Листинг 1.156: MSVC

```

_a$ = 8 ; size = 4
_f     PROC
mov    eax, DWORD PTR _a$[esp-4]
sub    eax, 0
je     SHORT $LN4@f
sub    eax, 1
je     SHORT $LN3@f
sub    eax, 1
je     SHORT $LN2@f

```

<sup>92</sup>Локальные переменные в стеке с префиксом `tv` — так MSVC называет внутренние переменные для своих нужд

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

mov     DWORD PTR _a$[esp-4], OFFSET $SG791 ; 'something unknown', 0aH,
00H
jmp     _printf
$LN2@f:
mov     DWORD PTR _a$[esp-4], OFFSET $SG789 ; 'two', 0aH, 00H
jmp     _printf
$LN3@f:
mov     DWORD PTR _a$[esp-4], OFFSET $SG787 ; 'one', 0aH, 00H
jmp     _printf
$LN4@f:
mov     DWORD PTR _a$[esp-4], OFFSET $SG785 ; 'zero', 0aH, 00H
jmp     _printf
_f     ENDP

```

Вот здесь уже всё немного по-другому, причем не без грязных трюков.

Первое: а помещается в EAX и от него отнимается 0. Звучит абсурдно, но нужно это для того, чтобы проверить, 0 ли в EAX был до этого? Если да, то выставится флаг ZF (что означает, что результат вычитания 0 от числа стал 0) и первый условный переход JE (*Jump if Equal* или его синоним JZ — *Jump if Zero*) сработает на метку \$LN4@f, где выводится сообщение 'zero'. Если первый переход не сработал, от значения отнимается по единице, и если на какой-то стадии в результате образуется 0, то сработает соответствующий переход.

И в конце концов, если ни один из условных переходов не сработал, управление передается printf() со строковым аргументом 'something unknown'.

Второе: мы видим две, мягко говоря, необычные вещи: указатель на сообщение помещается в переменную *a*, и затем printf() вызывается не через CALL, а через JMP. Объяснение этому простое. Вызывающая функция заталкивает в стек некоторое значение и через CALL вызывает нашу функцию. CALL в свою очередь заталкивает в стек адрес возврата (**RA!**) и делает безусловный переход на адрес нашей функции. Наша функция в самом начале (да и в любом её месте, потому что в теле функции нет ни одной инструкции, которая меняет что-то в стеке или в ESP) имеет следующую разметку стека:

- ESP— хранится **RA!**
- ESP+4 — хранится значение *a*

С другой стороны, чтобы вызвать printf(), нам нужна почти такая же разметка стека, только в первом аргументе нужен указатель на строку. Что, собственно, этот код и делает.

Он заменяет свой первый аргумент на адрес строки, и затем передает управление printf(), как если бы вызвали не нашу функцию f(), а сразу printf(). printf() выводит некую строку на [stdout](#), затем исполняет инструкцию RET, которая из стека достает **RA!** и управление передается в ту функцию, которая вызывала f(), минуя при этом конец функции f().

Всё это возможно, потому что printf() вызывается в f() в самом конце. Всё это чем-то даже похоже на longjmp()<sup>93</sup>. И всё это, разумеется, сделано для экономии времени исполнения.

<sup>93</sup> [wikipedia](#)

---

Похожая ситуация с компилятором для ARM описана в секции «printf() с несколькими аргументами» (?? (стр. ??)).

## OllYDbg

Так как этот пример немного запутанный, попробуем оттрассировать его в OllYDbg.

OllYDbg может распознавать подобные switch()-конструкции, так что он добавляет полезные комментарии. EAX в начале равен 2, это входное значение функции:

The screenshot shows the OllYDbg interface with the following details:

- Assembly View:**
  - 00FF1000: MOV EAX, DWORD PTR SS:[ARG.1]
  - 00FF1004: SUB EAX, 0
  - 00FF1007: JZ SHORT 00FF1039
  - 00FF1009: DEC EAX
  - 00FF100A: JZ SHORT 00FF102B
  - 00FF100C: DEC EAX
  - 00FF100D: JZ SHORT 00FF101D
  - 00FF100F: MOV DWORD PTR SS:[ARG.1], OFFSET 00FF301
  - 00FF1017: JMP DWORD PTR DS:[&MSUCR100.printf]
  - 00FF101D: MOV DWORD PTR SS:[ARG.1], OFFSET 00FF301
  - 00FF1025: JMP DWORD PTR DS:[&MSUCR100.printf]
  - 00FF102B: MOV DWORD PTR SS:[ARG.1], OFFSET 00FF300
  - 00FF1033: JMP DWORD PTR DS:[&MSUCR100.printf]
  - 00FF1039: MOV DWORD PTR SS:[ARG.1], OFFSET 00FF300
  - 00FF1041: JMP DWORD PTR DS:[&MSUCR100.printf]
  - 00FF1047: CC
  - 00FF1048: CC
  - 00FF1049: CC
  - 00FF104A: CC
  - 00FF104B: CC
  - 00FF104C: CC
- Registers (MMX):**
  - EAX: 00000002
  - ECX: 6E494714 ASCII "H(\*"
  - EDX: 00000000
  - ESI: 00000000
  - ESP: 001EF84C
  - EBP: 001EF894
  - EIP: 00FF33A8 few.00FF33A8
- Switch Statement:**
  - Switch (cases 0..2, 4 ex
  - ASCII "something unknown"
  - ASCII "two", case 2 of
  - ASCII "one", case 1 of
  - ASCII "zero", case 0 of
- Registers (MMX) - Bottom:**
  - 001EF84C: 00FF1057 RETURN from f
  - 001EF850: 00000002
  - 001EF854: 00FF11CA RETURN from f
  - 001EF858: 00000001
  - 001EF85C: 002A4E68 hN\* ASCII "pN"
  - 001EF860: 002A4E68 hN\*
  - 001EF864: 466BAC00 ankF
  - 001EF868: 00000000
  - 001EF86C: 00000000
  - 001EF870: 7EFDE000
  - 001EF874: 00000000
  - 001EF878: 00000000
  - 001EF87C: 001EF864 d\*
  - 001EF880: D3393100 ySL
  - 001EF884: 001EF800 Printer to ne
- Hex dump:**
  - Address: 00FF3000 to 00FF30C0
  - Hex dump: 65 72 6F 0A 00 00 00 00 6F 6E 65 0A 00 00 00 00
  - ASCII (ANSI - Cy): 0 0 one 0 something unknown 0 0 4TuF7nKl 0 H\* hN\*

Рис. 1.42: OllYDbg: EAX содержит первый (и единственный) аргумент функции

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

0 отнимается от 2 в EAX. Конечно же, EAX всё ещё содержит 2. Но флаг ZF теперь 0, что означает, что последнее вычисленное значение не было нулевым:

**CPU - main thread, module few**

Address	Hex dump	ASCII (ANSI - Cy)
00FF3000	74 77 6F 0A 00 00 00 6F 6E 65 0A 00 00 00 00	one
00FF3010	67 20 75 6E 6E 6E 6F 77 6E 0A 00 00 00 00	two something unknown
00FF3020	FE FF FF FF 01 00 00 00 34 54 75 46 CB AB 8A B9	4TuTnKil
00FF3030	01 00 00 00 48 23 2A 00 68 4E 2A 00 00 00 00	H(* hN*

**Registers (MMX)**

EAX	00000002
ECX	6E494714 ASCII "H(*)"
EDX	00000000
EBX	00000000
ESP	001EF84C
EBP	001EF594
ESI	00000001
EDI	00FF33A8 few.00FF33A8
EIP	00FF1007 few.00FF1007
CS	002B 32bit 0(FFFFFFFF)
DS	002B 32bit 0(FFFFFFFF)
SS	002B 32bit 0(FFFFFFFF)
FS	0053 32bit 7EFD0000(FFF)
GS	002B 32bit 0(FFFFFFFF)
EFL	0000202 (NO,NB,NE,A,NS,PO,GE,G)

Jump is not taken  
Dest=few.00FF1039

Рис. 1.43: OllyDbg: SUB исполнилась

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

DEC исполнулась и EAX теперь содержит 1. Но 1 не ноль, так что флаг ZF всё ещё 0:

The screenshot displays the OllyDbg interface for the CPU - main thread, module few. The assembly window shows the following instructions:

```

00FF1000 $ 8B4424 04 MOV EAX, DWORD PTR SS:[ARG.1]
00FF1004 83E8 00 SUB EAX, 0
00FF1007 74 30 JZ SHORT 00FF1039
00FF1009 48 DEC EAX
00FF100A 74 1F JZ SHORT 00FF102B
00FF100C 48 DEC EAX
00FF100D 74 0E JZ SHORT 00FF101D
00FF100F C74424 04 18 MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3011
00FF1017 FF25 0020FF00 JMP DWORD PTR DS:[&MSUCR100.printf]
00FF101D > C74424 04 18 MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3011
00FF1025 FF25 0020FF00 JMP DWORD PTR DS:[&MSUCR100.printf]
00FF102B > C74424 04 08 MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3008
00FF1033 FF25 0020FF00 JMP DWORD PTR DS:[&MSUCR100.printf]
00FF1039 > C74424 04 00 MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3000
00FF1041 FF25 0020FF00 JMP DWORD PTR DS:[&MSUCR100.printf]
00FF1047 CC INT3
00FF1048 CC INT3
00FF1049 CC INT3
00FF104A CC INT3
00FF104B CC INT3
00FF104C CC INT3

```

The registers window shows the following values:

```

Registers (MMX)
EAX 00000001
ECX 6E494714 ASCII "H(*"
EDX 00000000
EBX 00000000
ESP 001EF34C
EBP 001EF394
ESI 00000001
EDI 00FF33A8 few.00FF33A8
EIP 00FF100A few.00FF100A
C 0 ES 002B 32bit 0(FFFFFFFF)
P 0 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 0 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 7EFD0000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr 00000000 ERROR_SUCCESS
EFL 00002002 (NO, NB, NE, A, NS, PO, GE, G

```

The dump window shows the memory contents starting with 'zero one' and 'something unknown'.

Рис. 1.44: OllyDbg: первая DEC исполнулась

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Следующая DEC исполнилась. EAX наконец 0 и флаг ZF выставлен, потому что результат — ноль:

The screenshot shows the CPU window in OllyDbg. The registers pane on the right shows EAX as 00000000. The instruction list on the left shows a jump instruction at 00FF100D: JZ SHORT 00FF101D. The status bar at the bottom indicates that the jump is taken to destination few.00FF101D.

Рис. 1.45: OllyDbg: вторая DEC исполнилась

OllyDbg показывает, что условный переход сейчас сработает.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Указатель на строку «two» сейчас будет записан в стек:

The screenshot displays the OlyDbg interface with the following components:

- Assembly View:** Shows assembly instructions for a switch statement. The current instruction is `JMP DWORD PTR DS:[&MSUCR100.printf]` at address `00FF101D`, which is the label for the "two" case. The stack frame information indicates `Inn=few.00FF3010, ASCII "two"` and `Stack [001EF850]=2`.
- Registers (MMX):** Shows the current instruction pointer (EIP) at `00FF101D`.
- Stack View:** Shows the current value at memory address `001EF850` is `001EF850`, which is the address of the "two" case label. Other stack addresses show values like `001EF851` containing `0`.

Рис. 1.46: OlyDbg: указатель на строку сейчас запишется на место первого аргумента

Обратите внимание: текущий аргумент функции это 2 и 2 прямо сейчас в стеке по адресу `0x001EF850`.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

MOV записывает указатель на строку по адресу 0x001EF850 (см. окно стека). Переход сработал. Это самая первая инструкция функции printf() в MSVCR100.DLL (этот пример был скомпилирован с опцией /MD):

The screenshot displays the OllyDbg interface for the main thread in the MSVCR100 module. The CPU window shows assembly instructions for the function printf. The registers window shows the state of the CPU registers, with EAX set to 0 and ESP pointing to 001EF84C. The stack window shows the current instruction at address 00FF3000 and the next instruction at 00FF3010. The ASCII dump window shows the string 'H(\*' starting at address 001EF850.

Рис. 1.47: OllyDbg: первая инструкция в printf() в MSVCR100.DLL

Теперь printf() считает строку на 0x00FF3010 как свой единственный аргумент и выводит строку.



Это самая последняя инструкция функции printf():

The screenshot displays the CPU window for the main thread in module MSVCRT100. The instruction list shows the final instruction of the printf function: 6E445610: E8 7EB3FAFF CALL 6E3F0995 RETN. The registers window shows EIP at 6E445617. The stack window shows the top of the stack at 001EF84C containing the string 'two'.

Рис. 1.48: OllyDbg: последняя инструкция в printf() в MSVCRT100.DLL

Строка «two» была только что выведена в консоли.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Нажмем F7 или F8 (сделать шаг, не входя в функцию) и вернемся...нет, не в функцию f() но в main():

The screenshot shows the OllyDbg interface with the following details:

- Assembly Window:**
  - 00FF1048 CC INT3
  - 00FF1049 CC INT3
  - 00FF104A CC INT3
  - 00FF104B CC INT3
  - 00FF104C CC INT3
  - 00FF104D CC INT3
  - 00FF104E CC INT3
  - 00FF104F CC INT3
  - 00FF1050 6A 02 PUSH 2
  - 00FF1052 E8 A9FFFFFF CALL 00FF1000
  - 00FF1057 83C4 04 ADD ESP,4
  - 00FF105A 33C0 XOR EAX,EAX
  - 00FF105C C3 RETN
  - 00FF105D 68 2014FF00 PUSH 00FF142A
  - 00FF1062 E8 86030000 CALL 00FF13ED
  - 00FF1067 A1 7430FF00 MOV EAX, DWORD PTR DS:[0FF3074]
  - 00FF106C C70424 6430FF MOV DWORD PTR SS:[LOCAL\_0], OFFSET 00FF3064
  - 00FF1073 FF35 7030FF00 PUSH DWORD PTR DS:[0FF3070]
  - 00FF1079 A3 6430FF00 MOV DWORD PTR DS:[0FF3064], EAX
  - 00FF107E 68 5430FF00 PUSH OFFSET 00FF3054
  - 00FF1083 68 5830FF00 PUSH OFFSET 00FF3058
- Registers (MMX):**
  - EAX: 00000004
  - ECX: 6E445617 MSUCR100.6E445617
  - EDX: 0009DC88
  - EBX: 00000000
  - ESP: 001EF958 PTR to ASCII "two"
  - EBP: 001EF944
  - ESI: 00000001
  - EDI: 00FF33A8 few.00FF33A8
  - EIP: 00FF1057 few.00FF1057
- Stack:**
  - 00FF3000: 55 72 6F 0A 00 00 00 6F 6E 65 0A 00 00 00 00
  - 00FF3010: 74 77 6F 0A 00 00 00 73 6F 6D 65 74 68 69 6E
  - 00FF3020: 67 20 75 6E 6B 6E 6F 77 6E 0A 00 00 FF FF FF FF
  - 00FF3030: FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00
  - 00FF3040: FE FF FF 01 00 00 00 34 54 75 46 CB AB 8A B9
  - 00FF3050: 01 00 00 00 43 23 2A 00 68 4E 2A 00 00 00 00 00
  - 00FF3060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - 00FF3070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - 00FF3080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - 00FF3090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - 00FF30A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - 00FF30B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - 00FF30C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
- ASCII (ANSI - Cy):**
  - 001EF854: 00FF11CA RETURN from fe
  - 001EF858: 00000001 0
  - 001EF85C: 002A4E68 hN\*
  - 001EF860: 002A2348 H\*
  - 001EF864: 466BAC00 amkF
  - 001EF868: 00000000
  - 001EF86C: 00000000
  - 001EF870: 7EFD0000
  - 001EF874: 00000000
  - 001EF878: 00000000
  - 001EF87C: 001EF864 d\*
  - 001EF880: 03389310 b3\*
  - 001EF884: 001EF800 c\*
  - 001EF888: 00FE1619 Pointer to next handler

Рис. 1.49: OllyDbg: возврат в main()

Да, это прямой переход из внутренностей printf() в main(). Потому как **RA!** в стеке указывает не на какое-то место в функции f() а в main(). И CALL 0x00FF1000 это инструкция вызывающая функцию f().

### ARM: Оптимизирующий Keil 6/2013 (Режим ARM)

```
.text:0000014C          f1:
.text:0000014C 00 00 50 E3    CMP     R0, #0
.text:00000150 13 0E 8F 02    ADREQ  R0, aZero ; "zero\n"
.text:00000154 05 00 00 0A    BEQ    loc_170
.text:00000158 01 00 50 E3    CMP     R0, #1
.text:0000015C 4B 0F 8F 02    ADREQ  R0, aOne ; "one\n"
.text:00000160 02 00 00 0A    BEQ    loc_170
.text:00000164 02 00 50 E3    CMP     R0, #2
.text:00000168 4A 0F 8F 12    ADRNE  R0, aSomethingUnkno ; "something
          unknown\n"
.text:0000016C 4E 0F 8F 02    ADREQ  R0, aTwo ; "two\n"
.text:00000170
.text:00000170          loc_170: ; CODE XREF: f1+8
.text:00000170          ; f1+14
.text:00000170 78 18 00 EA    B      __2printf
```

Мы снова не сможем сказать, глядя на этот код, был ли в оригинальном исходном коде switch() либо же несколько операторов if().

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Так или иначе, мы снова видим здесь инструкции с предикатами, например, `ADREQ` (*Equal*), которая будет исполняться только если  $R0 = 0$ , и тогда в  $R0$  будет загружен адрес строки `«zero\n»`.

Следующая инструкция **BEQ!** перенаправит исполнение на `loc_170`, если  $R0 = 0$ .

Кстати, наблюдательный читатель может спросить, сработает ли **BEQ!** нормально, ведь `ADREQ` перед ним уже заполнила регистр  $R0$  чем-то другим?

Сработает, потому что **BEQ!** проверяет флаги, установленные инструкцией `CMP`, а `ADREQ` флаги никак не модифицирует.

Далее всё просто и знакомо. Вызов `printf()` один, и в самом конце, мы уже рассматривали подобный трюк (?? (стр. ??)). К вызову функции `printf()` в конце ведут три пути.

Последняя инструкция `CMP R0, #2` здесь нужна, чтобы узнать  $a = 2$  или нет.

Если это не так, то при помощи `ADRNE` (*Not Equal*) в  $R0$  будет загружен указатель на строку `«something unknown\n»`, ведь  $a$  уже было проверено на 0 и 1 до этого, и здесь  $a$  точно не попадает под эти константы.

Ну а если  $R0 = 2$ , в  $R0$  будет загружен указатель на строку `«two\n»` при помощи инструкции `ADREQ`.

### ARM: Оптимизирующий Keil 6/2013 (Режим Thumb)

<code>.text:000000D4</code>	<code>f1:</code>	
<code>.text:000000D4 10 B5</code>	<code>PUSH</code>	<code>{R4,LR}</code>
<code>.text:000000D6 00 28</code>	<code>CMP</code>	<code>R0, #0</code>
<code>.text:000000D8 05 D0</code>	<code>BEQ</code>	<code>zero_case</code>
<code>.text:000000DA 01 28</code>	<code>CMP</code>	<code>R0, #1</code>
<code>.text:000000DC 05 D0</code>	<code>BEQ</code>	<code>one_case</code>
<code>.text:000000DE 02 28</code>	<code>CMP</code>	<code>R0, #2</code>
<code>.text:000000E0 05 D0</code>	<code>BEQ</code>	<code>two_case</code>
<code>.text:000000E2 91 A0</code>	<code>ADR</code>	<code>R0, aSomethingUnkno ; "something</code>
<code>.text:000000E4 04 E0</code>	<code>B</code>	<code>default_case</code>
<code>.text:000000E6</code>	<code>zero_case: ; CODE XREF: f1+4</code>	
<code>.text:000000E6 95 A0</code>	<code>ADR</code>	<code>R0, aZero ; "zero\n"</code>
<code>.text:000000E8 02 E0</code>	<code>B</code>	<code>default_case</code>
<code>.text:000000EA</code>	<code>one_case: ; CODE XREF: f1+8</code>	
<code>.text:000000EA 96 A0</code>	<code>ADR</code>	<code>R0, aOne ; "one\n"</code>
<code>.text:000000EC 00 E0</code>	<code>B</code>	<code>default_case</code>
<code>.text:000000EE</code>	<code>two_case: ; CODE XREF: f1+C</code>	
<code>.text:000000EE 97 A0</code>	<code>ADR</code>	<code>R0, aTwo ; "two\n"</code>
<code>.text:000000F0</code>	<code>default_case ; CODE XREF: f1+10</code>	
<code>.text:000000F0</code>		<code>; f1+14</code>
<code>.text:000000F0 06 F0 7E F8</code>	<code>BL</code>	<code>__2printf</code>
<code>.text:000000F4 10 BD</code>	<code>POP</code>	<code>{R4,PC}</code>

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Как уже было отмечено, в Thumb-режиме нет возможности добавлять условные предикаты к большинству инструкций, так что Thumb-код вышел похожим на код x86 в стиле **CISC!**, вполне понятный.

#### ARM64: Неоптимизирующий GCC (Linaro) 4.9

```
.LC12:
.string "zero"
.LC13:
.string "one"
.LC14:
.string "two"
.LC15:
.string "something unknown"
f12:
    stp    x29, x30, [sp, -32]!
    add    x29, sp, 0
    str    w0, [x29,28]
    ldr    w0, [x29,28]
    cmp    w0, 1
    beq    .L34
    cmp    w0, 2
    beq    .L35
    cmp    w0, wzr
    bne    .L38           ; переход на метку по умолчанию
    adrp   x0, .LC12      ; "zero"
    add    x0, x0, :lo12:.LC12
    bl     puts
    b     .L32
.L34:
    adrp   x0, .LC13      ; "one"
    add    x0, x0, :lo12:.LC13
    bl     puts
    b     .L32
.L35:
    adrp   x0, .LC14      ; "two"
    add    x0, x0, :lo12:.LC14
    bl     puts
    b     .L32
.L38:
    adrp   x0, .LC15      ; "something unknown"
    add    x0, x0, :lo12:.LC15
    bl     puts
    nop
.L32:
    ldp    x29, x30, [sp], 32
    ret
```

Входное значение имеет тип *int*, поэтому для него используется регистр W0, а не целая часть регистра X0.

Указатели на строки передаются в puts ( ) при помощи пары инструкций ADRP/ADD, как было показано в примере «Hello, world!»: ?? (стр. ??).

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

**ARM64: Оптимизирующий GCC (Linaro) 4.9**

```

f12:
    cmp    w0, 1
    beq    .L31
    cmp    w0, 2
    beq    .L32
    cbz    w0, .L35
; метка по умолчанию
    adrp   x0, .LC15          ; "something unknown"
    add    x0, x0, :lo12:.LC15
    b      puts
.L35:
    adrp   x0, .LC12          ; "zero"
    add    x0, x0, :lo12:.LC12
    b      puts
.L32:
    adrp   x0, .LC14          ; "two"
    add    x0, x0, :lo12:.LC14
    b      puts
.L31:
    adrp   x0, .LC13          ; "one"
    add    x0, x0, :lo12:.LC13
    b      puts

```

Фрагмент кода более оптимизированный. Инструкция CBZ (*Compare and Branch on Zero* — сравнить и перейти если ноль) совершает переход если W0 ноль. Здесь также прямой переход на puts() вместо вызова, как уже было описано: ?? (стр. ??).

**MIPS**

Листинг 1.157: Оптимизирующий GCC 4.4.5 (IDA)

```

f:
    lui    $gp, (__gnu_local_gp >> 16)
; это 1?
    li     $v0, 1
    beq    $a0, $v0, loc_60
    la     $gp, (__gnu_local_gp & 0xFFFF) ; branch delay slot
; это 2?
    li     $v0, 2
    beq    $a0, $v0, loc_4C
    or     $at, $zero ; branch delay slot, NOP
; перейти, если не равно 0:
    bnez   $a0, loc_38
    or     $at, $zero ; branch delay slot, NOP
; случай нуля:
    lui    $a0, ($LC0 >> 16) # "zero"
    lw     $t9, (puts & 0xFFFF)($gp)
    or     $at, $zero ; load delay slot, NOP
    jr     $t9 ; branch delay slot, NOP
    la     $a0, ($LC0 & 0xFFFF) # "zero" ; branch delay slot

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

loc_38:                                # CODE XREF: f+1C
        lui    $a0, ($LC3 >> 16) # "something unknown"
        lw     $t9, (puts & 0xFFFF)($gp)
        or     $at, $zero ; load delay slot, NOP
        jr     $t9
        la     $a0, ($LC3 & 0xFFFF) # "something unknown" ; branch
        delay slot

loc_4C:                                # CODE XREF: f+14
        lui    $a0, ($LC2 >> 16) # "two"
        lw     $t9, (puts & 0xFFFF)($gp)
        or     $at, $zero ; load delay slot, NOP
        jr     $t9
        la     $a0, ($LC2 & 0xFFFF) # "two" ; branch delay slot

loc_60:                                # CODE XREF: f+8
        lui    $a0, ($LC1 >> 16) # "one"
        lw     $t9, (puts & 0xFFFF)($gp)
        or     $at, $zero ; load delay slot, NOP
        jr     $t9
        la     $a0, ($LC1 & 0xFFFF) # "one" ; branch delay slot

```

Функция всегда заканчивается вызовом `puts()`, так что здесь мы видим переход на `puts()` (JR: «Jump Register») вместо перехода с сохранением **RA!** («jump and link»).

Мы говорили об этом ранее: ?? (стр. ??).

Мы также часто видим NOP-инструкции после LW. Это «load delay slot»: ещё один *delay slot* в MIPS. Инструкция после LW может исполняться в тот момент, когда LW загружает значение из памяти.

Впрочем, следующая инструкция не должна использовать результат LW.

Современные MIPS-процессоры ждут, если следующая инструкция использует результат LW, так что всё это уже устарело, но GCC всё еще добавляет NOP-ы для более старых процессоров.

Вообще, это можно игнорировать.

## Вывод

Оператор `switch()` с малым количеством вариантов трудно отличим от применения конструкции `if/else`: листинг.??.

### 1.21.2. И если много

Если ветвлений слишком много, то генерировать слишком длинный код с многочисленными JE/JNE уже не так удобно.

```
#include <stdio.h>
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

void f (int a)
{
    switch (a)
    {
        case 0: printf ("zero\n"); break;
        case 1: printf ("one\n"); break;
        case 2: printf ("two\n"); break;
        case 3: printf ("three\n"); break;
        case 4: printf ("four\n"); break;
        default: printf ("something unknown\n"); break;
    };
};

int main()
{
    f (2); // test
};

```

## x86

### Неоптимизирующий MSVC

Рассмотрим пример, скомпилированный в (MSVC 2010):

Листинг 1.158: MSVC 2010

```

tv64 = -4 ; size = 4
_a$ = 8 ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _a$[ebp]
    mov     DWORD PTR tv64[ebp], eax
    cmp     DWORD PTR tv64[ebp], 4
    ja     SHORT $LN1@f
    mov     ecx, DWORD PTR tv64[ebp]
    jmp     DWORD PTR $LN11@f[ecx*4]
$LN6@f:
    push    OFFSET $SG739 ; 'zero', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
$LN5@f:
    push    OFFSET $SG741 ; 'one', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
$LN4@f:
    push    OFFSET $SG743 ; 'two', 0aH, 00H
    call    _printf
    add     esp, 4

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    jmp     SHORT $LN9@f
$LN3@f:
    push   OFFSET $SG745 ; 'three', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN2@f:
    push   OFFSET $SG747 ; 'four', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN1@f:
    push   OFFSET $SG749 ; 'something unknown', 0aH, 00H
    call   _printf
    add    esp, 4
$LN9@f:
    mov    esp, ebp
    pop    ebp
    ret    0
    nrad   2 ; выровнять следующую метку
$LN11@f:
    DD     $LN6@f ; 0
    DD     $LN5@f ; 1
    DD     $LN4@f ; 2
    DD     $LN3@f ; 3
    DD     $LN2@f ; 4
_f      ENDP

```

Здесь происходит следующее: в теле функции есть набор вызовов `printf()` с разными аргументами. Все они имеют, конечно же, адреса, а также внутренние символические метки, которые присвоил им компилятор. Также все эти метки указываются во внутренней таблице `$LN11@f`.

В начале функции, если  $a$  больше 4, то сразу происходит переход на метку `$LN1@f`, где вызывается `printf()` с аргументом `'something unknown'`.

А если  $a$  меньше или равно 4, то это значение умножается на 4 и прибавляется адрес таблицы с переходами (`$LN11@f`). Таким образом, получается адрес внутри таблицы, где лежит нужный адрес внутри тела функции. Например, возьмем  $a$  равным 2.  $2 * 4 = 8$  (ведь все элементы таблицы — это адреса внутри 32-битного процесса, таким образом, каждый элемент занимает 4 байта). 8 прибавить к `$LN11@f` — это будет элемент таблицы, где лежит `$LN4@f`. `JMP` вытаскивает из таблицы адрес `$LN4@f` и делает безусловный переход туда.

Эта таблица иногда называется *jump table* или *branch table*<sup>94</sup>.

А там вызывается `printf()` с аргументом `'two'`. Дословно, инструкция `jmp DWORD PTR $LN11@f[ecx*4]` означает *перейти по DWORD, который лежит по адресу \$LN11@f + ecx \* 4*.

`nrad (?? (стр. ??))` это макрос ассемблера, выравнивающий начало таблицы,

<sup>94</sup>Сам метод раньше назывался *computed GOTO* В ранних версиях Фортрана: [wikipedia](#). Не очень-то и полезно в наше время, но каков термин!



---

чтобы она располагалась по адресу кратному 4 (или 16). Это нужно для того, чтобы процессор мог эффективнее загружать 32-битные значения из памяти через шину с памятью, кэш-память, и т. д.

## OllYDbg

Попробуем этот пример в OllYDbg. Входное значение функции (2) загружается в EAX:

The screenshot displays the OllYDbg interface for the CPU - main thread, module lot. The assembly window shows the following instructions:

```

010B1000 55 PUSH EBP
010B1001 8BEC MOV EBP,ESP
010B1003 51 PUSH ECX
010B1004 8B45 08 MOV EAX,DWORD PTR SS:[EBP+8]
010B1007 8945 FC MOV DWORD PTR SS:[EBP-4],EAX
010B100A 837D FC 04 CMP DWORD PTR SS:[EBP-4],4
010B100E 77 5A JA SHORT 010B106A
010B1010 8B4D FC MOV ECX,DWORD PTR SS:[EBP-4]
010B1013 FF248D 7C100 CALL DWORD PTR DS:[ECX*4+10B107C]
010B101A 68 00300000 PUSH OFFSET 010B3000
010B101F FF15 00200000 CALL DWORD PTR DS:[&MSUCR100.printf]
010B1025 83C4 04 ADD ESP,4
010B1028 EB 4E JMP SHORT 010B1078
010B102A 68 00300000 PUSH OFFSET 010B3000
010B102F FF15 00200000 CALL DWORD PTR DS:[&MSUCR100.printf]
010B1035 83C4 04 ADD ESP,4
010B1038 EB 3E JMP SHORT 010B1078
010B103A 68 00300000 PUSH OFFSET 010B3010
010B103F FF15 00200000 CALL DWORD PTR DS:[&MSUCR100.printf]
010B1045 83C4 04 ADD ESP,4
010B1048 EB 2E JMP SHORT 010B1078
  
```

The Registers (MMX) window shows the following values:

```

EAX 00000002
ECX 6E494714 MSUCR100.__initenv
EDX 00000000
EBX 00000000
ESP 003CFD88
EBP 003CFD8C
ESI 00000001
EDI 010B33B8 lot.010B33B8
EIP 010B1007 lot.010B1007
  
```

The Stack window shows the following data:

```

EAX=2
Stack [003CFD88]=6E494714 (MSUCR100.__initenv)
  
```

The Hex dump window shows the following data:

```

Address Hex dump ASCII (ANSI - Cy)
010B3000 74 65 72 6F 0A 00 00 00 00 00 00 00 00 00 00 zero one
010B3010 74 77 6F 0A 00 00 00 00 74 65 72 65 65 0A 00 two three
010B3020 66 6F 75 72 0A 00 00 00 73 6F 6D 65 74 68 63 four someth in
010B3030 67 20 75 6E 6B 6E 6F 77 6E 0A 00 00 FF FF FF FF g unknown
010B3040 FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00
010B3050 FE FF FF FF 01 00 00 00 9A E2 68 1D 65 1D 97 E2
010B3060 01 00 00 00 48 23 03 00 68 4E 03 00 00 00 00 00
010B3070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
010B3080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
010B3090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
010B30A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
010B30B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
010B30C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  
```

Рис. 1.50: OllYDbg: входное значение функции загружено в EAX

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



Здесь мы видим jumtable:

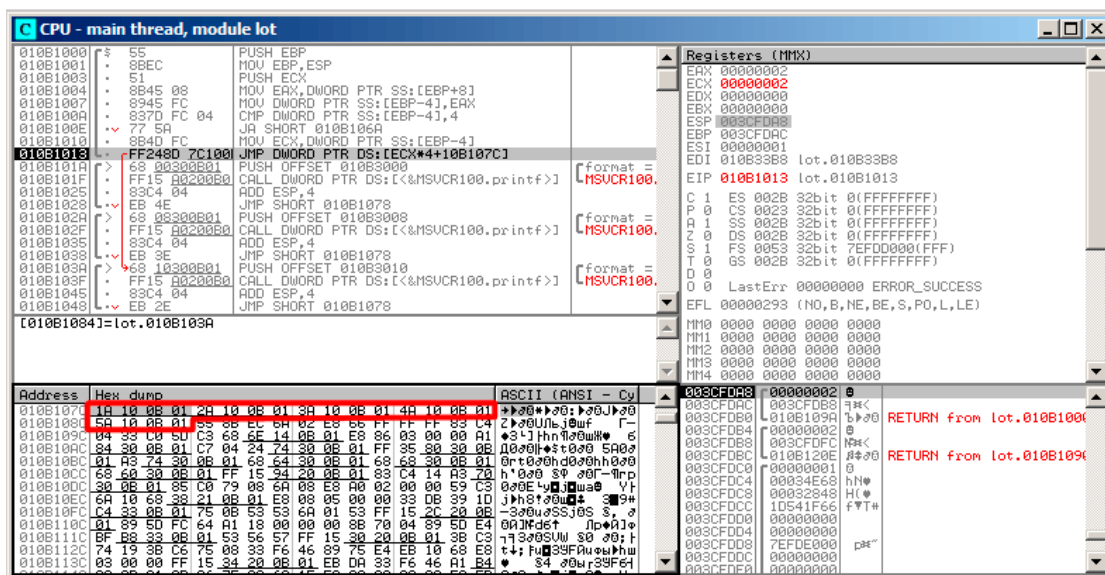


Рис. 1.52: OllyDbg: вычисляем адрес для перехода используя jumtable

Кстати, щелкнем по «Follow in Dump» → «Address constant», так что теперь *jumtable* видна в окне данных.

Это 5 32-битных значений<sup>95</sup>. ECX сейчас содержит 2, так что третий элемент (может индексироваться как 2<sup>96</sup>) таблицы будет использован. Кстати, можно также щелкнуть «Follow in Dump» → «Memory address» и OllyDbg покажет элемент, который сейчас адресуется в инструкции JMP. Это 0x010B103A.

<sup>95</sup>Они подчеркнуты в OllyDbg, потому что это также и FIXUP-ы: ?? (стр. ??), мы вернемся к ним позже

<sup>96</sup>Об индексации, см. также: ?? (стр. ??)

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Переход сработал и мы теперь на 0x010B103A: сейчас будет исполнен код, выводящий строку «two»:

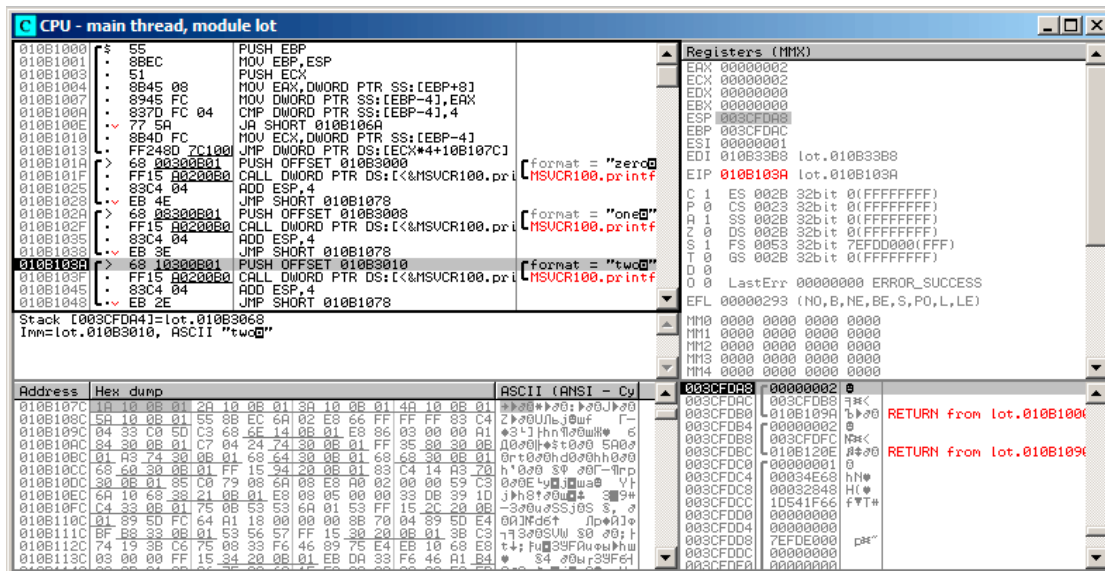


Рис. 1.53: OllyDbg: теперь мы на соответствующей метке case:

## Неоптимизирующий GCC

Посмотрим, что сгенерирует GCC 4.4.1:

Листинг 1.159: GCC 4.4.1

```

public f
f
proc near ; CODE XREF: main+10

var_18 = dword ptr -18h
arg_0 = dword ptr 8

push    ebp
mov     ebp, esp
sub     esp, 18h
cmp     [ebp+arg_0], 4
ja     short loc_8048444
mov     eax, [ebp+arg_0]
shl    eax, 2
mov     eax, ds:off_804855C[eax]
jmp     eax

loc_80483FE: ; DATA XREF: .rodata:off_804855C
mov     [esp+18h+var_18], offset aZero ; "zero"
call   _puts
jmp     short locret_8048450

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

loc_804840C: ; DATA XREF: .rodata:08048560
            mov     [esp+18h+var_18], offset aOne ; "one"
            call   _puts
            jmp    short locret_8048450

loc_804841A: ; DATA XREF: .rodata:08048564
            mov     [esp+18h+var_18], offset aTwo ; "two"
            call   _puts
            jmp    short locret_8048450

loc_8048428: ; DATA XREF: .rodata:08048568
            mov     [esp+18h+var_18], offset aThree ; "three"
            call   _puts
            jmp    short locret_8048450

loc_8048436: ; DATA XREF: .rodata:0804856C
            mov     [esp+18h+var_18], offset aFour ; "four"
            call   _puts
            jmp    short locret_8048450

loc_8048444: ; CODE XREF: f+A
            mov     [esp+18h+var_18], offset aSomethingUnkno ; "something unknown"
            call   _puts

locret_8048450: ; CODE XREF: f+26
                ; f+34...
            leave
            retn
f            endp

off_804855C dd offset loc_80483FE ; DATA XREF: f+12
            dd offset loc_804840C
            dd offset loc_804841A
            dd offset loc_8048428
            dd offset loc_8048436

```

Практически то же самое, за исключением мелкого нюанса: аргумент из `arg_0` умножается на 4 при помощи сдвига влево на 2 бита (это почти то же самое что и умножение на 4) (?? (стр. ??)). Затем адрес метки внутри функции берется из массива `off_804855C` и адресуется при помощи вычисленного индекса.

### ARM: Оптимизирующий Keil 6/2013 (Режим ARM)

Листинг 1.160: Оптимизирующий Keil 6/2013 (Режим ARM)

```

00000174          f2
00000174 05 00 50 E3    CMP     R0, #5           ; switch 5 cases
00000178 00 F1 8F 30    ADDCC  PC, PC, R0,LSL#2 ; switch jump
0000017C 0E 00 00 EA    B      default_case     ; jumptable 00000178
                default case
00000180

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

00000180          loc_180 ; CODE XREF: f2+4
00000180 03 00 00 EA      B      zero_case      ; jumptable 00000178 case 0

00000184
00000184          loc_184 ; CODE XREF: f2+4
00000184 04 00 00 EA      B      one_case       ; jumptable 00000178 case 1

00000188
00000188          loc_188 ; CODE XREF: f2+4
00000188 05 00 00 EA      B      two_case       ; jumptable 00000178 case 2

0000018C
0000018C          loc_18C ; CODE XREF: f2+4
0000018C 06 00 00 EA      B      three_case      ; jumptable 00000178 case 3

00000190
00000190          loc_190 ; CODE XREF: f2+4
00000190 07 00 00 EA      B      four_case      ; jumptable 00000178 case 4

00000194
00000194          zero_case ; CODE XREF: f2+4
00000194          ; f2:loc_180
00000194 EC 00 8F E2      ADR      R0, aZero      ; jumptable 00000178 case 0
00000198 06 00 00 EA      B      loc_1B8

0000019C
0000019C          one_case ; CODE XREF: f2+4
0000019C          ; f2:loc_184
0000019C EC 00 8F E2      ADR      R0, aOne      ; jumptable 00000178 case 1
000001A0 04 00 00 EA      B      loc_1B8

000001A4
000001A4          two_case ; CODE XREF: f2+4
000001A4          ; f2:loc_188
000001A4 01 0C 8F E2      ADR      R0, aTwo      ; jumptable 00000178 case 2
000001A8 02 00 00 EA      B      loc_1B8

000001AC
000001AC          three_case ; CODE XREF: f2+4
000001AC          ; f2:loc_18C
000001AC 01 0C 8F E2      ADR      R0, aThree    ; jumptable 00000178 case 3
000001B0 00 00 00 EA      B      loc_1B8

000001B4
000001B4          four_case ; CODE XREF: f2+4
000001B4          ; f2:loc_190
000001B4 01 0C 8F E2      ADR      R0, aFour     ; jumptable 00000178 case 4
000001B8
000001B8          loc_1B8 ; CODE XREF: f2+24
000001B8          ; f2+2C
000001B8 66 18 00 EA      B      __2printf

000001BC

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

000001BC	default_case ; CODE XREF: f2+4		
000001BC	; f2+8		
000001BC D4 00 8F E2	ADR	R0, aSomethingUnkno ; jumptable 00000178	
	default_case		
000001C0 FC FF FF EA	B	loc_1B8	

В этом коде используется та особенность режима ARM, что все инструкции в этом режиме имеют фиксированную длину 4 байта.

Итак, не будем забывать, что максимальное значение для  $a$  это 4: всё что выше, должно вызвать вывод строки «*something unknown*».

Самая первая инструкция CMP R0, #5 сравнивает входное значение в  $a$  с 5.

<sup>97</sup> Следующая инструкция ADDCC PC, PC, R0, LSL#2 сработает только в случае если  $R0 < 5$  ( $CC=Carry\ clear / Less\ than$ ). Следовательно, если ADDCC не сработает (это случай с  $R0 \geq 5$ ), выполнится переход на метку *default\_case*.

Но если  $R0 < 5$  и ADDCC сработает, то произойдет следующее.

Значение в R0 умножается на 4. Фактически, LSL#2 в суффиксе инструкции означает «сдвиг влево на 2 бита».

Но как будет видно позже (?? (стр. ??)) в секции «Сдвиги», сдвиг влево на 2 бита, это эквивалентно его умножению на 4.

Затем полученное  $R0 * 4$  прибавляется к текущему значению **PC!**, совершая, таким образом, переход на одну из расположенных ниже инструкций B (*Branch*).

На момент исполнения ADDCC, содержимое **PC!** на 8 байт больше (0x180), чем адрес по которому расположена сама инструкция ADDCC (0x178), либо, говоря иным языком, на 2 инструкции больше.

Это связано с работой конвейера процессора ARM: пока исполняется инструкция ADDCC, процессор уже начинает обрабатывать инструкцию после следующей, поэтому **PC!** указывает туда. Этот факт нужно запомнить.

Если  $a = 0$ , тогда к **PC!** ничего не будет прибавлено и в **PC!** запишется актуальный на тот момент **PC!** (который больше на 8) и произойдет переход на метку *loc\_180*. Это на 8 байт дальше места, где находится инструкция ADDCC.

Если  $a = 1$ , тогда в **PC!** запишется  $PC + 8 + a * 4 = PC + 8 + 1 * 4 = PC + 12 = 0x184$ . Это адрес метки *loc\_184*.

При каждой добавленной к  $a$  единице итоговый **PC!** увеличивается на 4.

4 это длина инструкции в режиме ARM и одновременно с этим, длина каждой инструкции B, их здесь следует 5 в ряд.

Каждая из этих пяти инструкций B передает управление дальше, где собственно и происходит то, что запрограммировано в операторе *switch()*. Там происходит загрузка указателя на свою строку, и т. д.

<sup>97</sup> ADD—складывание чисел



**ARM: Оптимизирующий Keil 6/2013 (Режим Thumb)**

Листинг 1.161: Оптимизирующий Keil 6/2013 (Режим Thumb)

```

000000F6                                EXPORT f2
000000F6                                f2
000000F6 10 B5                          PUSH   {R4,LR}
000000F8 03 00                          MOVS   R3, R0
000000FA 06 F0 69 F8                    BL     __ARM_common_switch8_thumb ; switch 6
cases
000000FE 05                              DCB 5
000000FF 04 06 08 0A 0C 10             DCB 4, 6, 8, 0xA, 0xC, 0x10 ; jump table for
switch statement
00000105 00                              ALIGN 2
00000106
00000106                                zero_case ; CODE XREF: f2+4
00000106 8D A0                          ADR   R0, aZero ; jumtable 000000FA case 0
00000108 06 E0                          B     loc_118

0000010A
0000010A                                one_case ; CODE XREF: f2+4
0000010A 8E A0                          ADR   R0, aOne ; jumtable 000000FA case 1
0000010C 04 E0                          B     loc_118

0000010E
0000010E                                two_case ; CODE XREF: f2+4
0000010E 8F A0                          ADR   R0, aTwo ; jumtable 000000FA case 2
00000110 02 E0                          B     loc_118

00000112
00000112                                three_case ; CODE XREF: f2+4
00000112 90 A0                          ADR   R0, aThree ; jumtable 000000FA case 3
00000114 00 E0                          B     loc_118

00000116
00000116                                four_case ; CODE XREF: f2+4
00000116 91 A0                          ADR   R0, aFour ; jumtable 000000FA case 4
00000118
00000118                                loc_118 ; CODE XREF: f2+12
00000118                                ; f2+16
00000118 06 F0 6A F8                    BL     __2printf
0000011C 10 BD                          POP   {R4,PC}

0000011E
0000011E                                default_case ; CODE XREF: f2+4
0000011E 82 A0                          ADR   R0, aSomethingUnkno ; jumtable
000000FA default case
00000120 FA E7                          B     loc_118

000061D0                                EXPORT __ARM_common_switch8_thumb
000061D0                                __ARM_common_switch8_thumb ; CODE XREF:
example6_f2+4
000061D0 78 47                          BX    PC

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

000061D2 00 00                ALIGN 4
000061D2                ; End of function __ARM_common_switch8_thumb
000061D2
000061D4                __32__ARM_common_switch8_thumb ; CODE XREF:
                        __ARM_common_switch8_thumb
000061D4 01 C0 5E E5          LDRB    R12, [LR,#-1]
000061D8 0C 00 53 E1          CMP     R3, R12
000061DC 0C 30 DE 27          LDRCSB R3, [LR,R12]
000061E0 03 30 DE 37          LDRCCB R3, [LR,R3]
000061E4 83 C0 8E E0          ADD    R12, LR, R3,LSL#1
000061E8 1C FF 2F E1          BX     R12
000061E8                ; End of function __32__ARM_common_switch8_thumb

```

В режимах Thumb и Thumb-2 уже нельзя надеяться на то, что все инструкции имеют одну длину.

Можно даже сказать, что в этих режимах инструкции переменной длины, как в x86.

Так что здесь добавляется специальная таблица, содержащая информацию о том, как много вариантов здесь, не включая варианта по умолчанию, и смещения, для каждого варианта. Каждое смещение кодирует метку, куда нужно передать управление в соответствующем случае.

Для того чтобы работать с таблицей и совершить переход, вызывается служебная функция

`__ARM_common_switch8_thumb`. Она начинается с инструкции `BX PC`, чья функция — переключить процессор в ARM-режим.

Далее функция, работающая с таблицей. Она слишком сложная для рассмотрения в данном месте, так что пропустим это.

Но можно отметить, что эта функция использует регистр **LR!** как указатель на таблицу.

Действительно, после вызова этой функции, в **LR!** был записан адрес после инструкции

`BL __ARM_common_switch8_thumb`, а там как раз и начинается таблица.

Ещё можно отметить, что код для этого выделен в отдельную функцию для того, чтобы не нужно было каждый раз генерировать точно такой же фрагмент кода для каждого выражения `switch()`.

**IDA!** распознала эту служебную функцию и таблицу автоматически дописала комментарии к меткам вроде `jumptable 000000FA case 0`.

## MIPS

Листинг 1.162: Оптимизирующий GCC 4.4.5 (IDA)

```

f:
    lui    $gp, (__gnu_local_gp >> 16)

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; перейти на loc_24, если входное значение меньше 5:
    sltiu    $v0, $a0, 5
    bnez     $v0, loc_24
    la       $gp, (__gnu_local_gp & 0xFFFF) ; branch delay slot
; входное значение больше или равно 5
; вывести "something unknown" и закончить:
    lui     $a0, ($LC5 >> 16) # "something unknown"
    lw      $t9, (puts & 0xFFFF)($gp)
    or      $at, $zero ; NOP
    jr      $t9
    la      $a0, ($LC5 & 0xFFFF) # "something unknown" ; branch
delay slot

loc_24:                                # CODE XREF: f+8
; загрузить адрес таблицы переходов
; LA это псевдоинструкция, на самом деле здесь пара LUI и ADDIU:
    la      $v0, off_120
; умножить входное значение на 4:
    sll     $a0, 2
; прибавить умноженное значение к адресу таблицы:
    addu    $a0, $v0, $a0
; загрузить элемент из таблицы переходов:
    lw      $v0, 0($a0)
    or      $at, $zero ; NOP
; перейти по адресу, полученному из таблицы:
    jr      $v0
    or      $at, $zero ; branch delay slot, NOP

sub_44:                                # DATA XREF: .rodata:0000012C
; вывести "three" и закончить
    lui     $a0, ($LC3 >> 16) # "three"
    lw      $t9, (puts & 0xFFFF)($gp)
    or      $at, $zero ; NOP
    jr      $t9
    la      $a0, ($LC3 & 0xFFFF) # "three" ; branch delay slot

sub_58:                                # DATA XREF: .rodata:00000130
; вывести "four" и закончить
    lui     $a0, ($LC4 >> 16) # "four"
    lw      $t9, (puts & 0xFFFF)($gp)
    or      $at, $zero ; NOP
    jr      $t9
    la      $a0, ($LC4 & 0xFFFF) # "four" ; branch delay slot

sub_6C:                                # DATA XREF: .rodata:off_120
; вывести "zero" и закончить
    lui     $a0, ($LC0 >> 16) # "zero"
    lw      $t9, (puts & 0xFFFF)($gp)
    or      $at, $zero ; NOP
    jr      $t9
    la      $a0, ($LC0 & 0xFFFF) # "zero" ; branch delay slot

sub_80:                                # DATA XREF: .rodata:00000124
; вывести "one" и закончить

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        lui    $a0, ($LC1 >> 16) # "one"
        lw     $t9, (puts & 0xFFFF)($gp)
        or     $at, $zero ; NOP
        jr     $t9
        la     $a0, ($LC1 & 0xFFFF) # "one" ; branch delay slot

sub_94:                                # DATA XREF: .rodata:00000128
; вывести "two" и закончить
        lui    $a0, ($LC2 >> 16) # "two"
        lw     $t9, (puts & 0xFFFF)($gp)
        or     $at, $zero ; NOP
        jr     $t9
        la     $a0, ($LC2 & 0xFFFF) # "two" ; branch delay slot

; может быть размещено в секции .rodata:
off_120:    .word sub_6C
            .word sub_80
            .word sub_94
            .word sub_44
            .word sub_58

```

Новая для нас инструкция здесь это SLTIU («Set on Less Than Immediate Unsigned» — установить, если меньше чем значение, беззнаковое сравнение).

На самом деле, это то же что и SLTU («Set on Less Than Unsigned»), но «I» означает «immediate», т.е. число может быть задано в самой инструкции.

BNEZ это «Branch if Not Equal to Zero» (переход если не равно нулю).

Код очень похож на код для других **ISA!** SLL («Shift Word Left Logical» — логический сдвиг влево) совершает умножение на 4. MIPS всё-таки это 32-битный процессор, так что все адреса в таблице переходов (*jump table*) 32-битные.

## Вывод

Примерный скелет оператора *switch()*:

Листинг 1.163: x86

```

MOV REG, input
CMP REG, 4 ; максимальное количество меток
JA default
SHL REG, 2 ; найти элемент в таблице. сдвинуть на 3 бита в x64
MOV REG, jump_table[REG]
JMP REG

case1:
    ; делать что-то
    JMP exit
case2:
    ; делать что-то
    JMP exit
case3:
    ; делать что-то

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    JMP exit
case4:
    ; делать что-то
    JMP exit
case5:
    ; делать что-то
    JMP exit

default:
    ...

exit:
    ....

jump_table dd case1
            dd case2
            dd case3
            dd case4
            dd case5

```

Переход по адресу из таблицы переходов может быть также реализован такой инструкцией:

JMP jump\_table[REG\*4]. Или JMP jump\_table[REG\*8] в x64.

Таблица переходов (*jump table*) это просто массив указателей, как это будет вскоре описано: ?? (стр. ??).

### 1.21.3. Когда много case в одном блоке

Вот очень часто используемая конструкция: несколько case может быть использовано в одном блоке:

```

#include <stdio.h>

void f(int a)
{
    switch (a)
    {
        case 1:
        case 2:
        case 7:
        case 10:
            printf ("1, 2, 7, 10\n");
            break;

        case 3:
        case 4:
        case 5:
        case 6:
            printf ("3, 4, 5\n");
            break;

        case 8:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    case 9:
    case 20:
    case 21:
        printf ("8, 9, 21\n");
        break;
    case 22:
        printf ("22\n");
        break;
    default:
        printf ("default\n");
        break;
};

int main()
{
    f(4);
};

```

Слишком расточительно генерировать каждый блок для каждого случая, поэтому обычно генерируется каждый блок плюс некий диспетчер.

## MSVC

Листинг 1.164: Оптимизирующий MSVC 2010

```

1  $SG2798 DB      '1, 2, 7, 10', 0aH, 00H
2  $SG2800 DB      '3, 4, 5', 0aH, 00H
3  $SG2802 DB      '8, 9, 21', 0aH, 00H
4  $SG2804 DB      '22', 0aH, 00H
5  $SG2806 DB      'default', 0aH, 00H
6
7  _a$ = 8
8  _f      PROC
9          mov     eax, DWORD PTR _a$[esp-4]
10         dec     eax
11         cmp     eax, 21
12         ja      SHORT $LN1@f
13         movzx   eax, BYTE PTR $LN10@f[eax]
14         jmp     DWORD PTR $LN11@f[eax*4]
15 $LN5@f:
16         mov     DWORD PTR _a$[esp-4], OFFSET $SG2798 ; '1, 2, 7, 10'
17         jmp     DWORD PTR __imp__printf
18 $LN4@f:
19         mov     DWORD PTR _a$[esp-4], OFFSET $SG2800 ; '3, 4, 5'
20         jmp     DWORD PTR __imp__printf
21 $LN3@f:
22         mov     DWORD PTR _a$[esp-4], OFFSET $SG2802 ; '8, 9, 21'
23         jmp     DWORD PTR __imp__printf
24 $LN2@f:
25         mov     DWORD PTR _a$[esp-4], OFFSET $SG2804 ; '22'
26         jmp     DWORD PTR __imp__printf
27 $LN1@f:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

28     mov     DWORD PTR _a$[esp-4], OFFSET $SG2806 ; 'default'
29     jmp     DWORD PTR __imp__printf
30     npad   2 ; выровнять таблицу $LN11@f по 16-байтной границе
31 $LN11@f:
32     DD     $LN5@f ; вывести '1, 2, 7, 10'
33     DD     $LN4@f ; вывести '3, 4, 5'
34     DD     $LN3@f ; вывести '8, 9, 21'
35     DD     $LN2@f ; вывести '22'
36     DD     $LN1@f ; вывести 'default'
37 $LN10@f:
38     DB     0 ; a=1
39     DB     0 ; a=2
40     DB     1 ; a=3
41     DB     1 ; a=4
42     DB     1 ; a=5
43     DB     1 ; a=6
44     DB     0 ; a=7
45     DB     2 ; a=8
46     DB     2 ; a=9
47     DB     0 ; a=10
48     DB     4 ; a=11
49     DB     4 ; a=12
50     DB     4 ; a=13
51     DB     4 ; a=14
52     DB     4 ; a=15
53     DB     4 ; a=16
54     DB     4 ; a=17
55     DB     4 ; a=18
56     DB     4 ; a=19
57     DB     2 ; a=20
58     DB     2 ; a=21
59     DB     3 ; a=22
60 _f     ENDP

```

Здесь видим две таблицы: первая таблица (\$LN10@f) это таблица индексов, а вторая таблица (\$LN11@f) это массив указателей на блоки.

В начале, входное значение используется как индекс в таблице индексов (строка 13).

Вот краткое описание значений в таблице: 0 это первый блок case (для значений 1, 2, 7, 10), 1 это второй (для значений 3, 4, 5), 2 это третий (для значений 8, 9, 21), 3 это четвертый (для значений 22), 4 это для блока по умолчанию.

Мы получаем индекс для второй таблицы указателей на блоки и переходим туда (строка 14).

Ещё нужно отметить то, что здесь нет случая для нулевого входного значения.

Поэтому мы видим инструкцию DEC на строке 10 и таблица начинается с  $a = 1$ . Потому что незачем выделять в таблице элемент для  $a = 0$ .

Это очень часто используемый шаблон.

В чем же экономия? Почему нельзя сделать так, как уже обсуждалось (?? (стр. ??)),

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

используя только одну таблицу, содержащую указатели на блоки? Причина в том, что элементы в таблице индексов занимают только по 8-битному байту, поэтому всё это более компактно.

## GCC

GCC делает так, как уже обсуждалось (?? (стр. ??)), используя просто таблицу указателей.

### ARM64: Оптимизирующий GCC 4.9.1

Во-первых, здесь нет кода, срабатывающего в случае если входное значение — 0, так что GCC пытается сделать таблицу переходов более компактной и начинает со случая, когда входное значение — 1.

GCC 4.9.1 для ARM64 использует даже более интересный трюк. Он может закодировать все смещения как 8-битные байты. Вспомним, что все инструкции в ARM64 имеют размер в 4 байта.

GCC также использует тот факт, что все смещения в моем крохотном примере находятся достаточно близко друг от друга.

Так что таблица переходов состоит из байт.

Листинг 1.165: Оптимизирующий GCC 4.9.1 ARM64

```
f14:
; входное значение в W0
    sub    w0, w0, #1
    cmp    w0, 21
; переход если меньше или равно (беззнаковое):
    bls    .L9
.L2:
; вывести "default":
    adrp   x0, .LC4
    add    x0, x0, :lo12:.LC4
    b      puts
.L9:
; загрузить адрес таблицы переходов в X1:
    adrp   x1, .L4
    add    x1, x1, :lo12:.L4
; W0=input_value-1
; загрузить байт из таблицы:
    ldrb   w0, [x1,w0,uxtw]
; загрузить адрес метки Lrtx:
    adr    x1, .Lrtx4
; умножить элемент из таблицы на 4 (сдвинув на 2 бита влево) и прибавить (или
    вычесть) к адресу Lrtx:
    add    x0, x1, w0, sxtb #2
; перейти на вычисленный адрес:
    br     x0
; эта метка указывает на сегмент кода (text):
.Lrtx4:
    .section      .rodata
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

; всё после выражения ".section" выделяется в сегменте только для чтения
(rodata):
.L4:
    .byte (.L3 - .Lrtx4) / 4    ; case 1
    .byte (.L3 - .Lrtx4) / 4    ; case 2
    .byte (.L5 - .Lrtx4) / 4    ; case 3
    .byte (.L5 - .Lrtx4) / 4    ; case 4
    .byte (.L5 - .Lrtx4) / 4    ; case 5
    .byte (.L5 - .Lrtx4) / 4    ; case 6
    .byte (.L3 - .Lrtx4) / 4    ; case 7
    .byte (.L6 - .Lrtx4) / 4    ; case 8
    .byte (.L6 - .Lrtx4) / 4    ; case 9
    .byte (.L3 - .Lrtx4) / 4    ; case 10
    .byte (.L2 - .Lrtx4) / 4    ; case 11
    .byte (.L2 - .Lrtx4) / 4    ; case 12
    .byte (.L2 - .Lrtx4) / 4    ; case 13
    .byte (.L2 - .Lrtx4) / 4    ; case 14
    .byte (.L2 - .Lrtx4) / 4    ; case 15
    .byte (.L2 - .Lrtx4) / 4    ; case 16
    .byte (.L2 - .Lrtx4) / 4    ; case 17
    .byte (.L2 - .Lrtx4) / 4    ; case 18
    .byte (.L2 - .Lrtx4) / 4    ; case 19
    .byte (.L6 - .Lrtx4) / 4    ; case 20
    .byte (.L6 - .Lrtx4) / 4    ; case 21
    .byte (.L7 - .Lrtx4) / 4    ; case 22
    .text
; всё после выражения ".text" выделяется в сегменте кода (text):
.L7:
; вывести "22"
    adrp    x0, .LC3
    add     x0, x0, :lo12:LC3
    b       puts
.L6:
; вывести "8, 9, 21"
    adrp    x0, .LC2
    add     x0, x0, :lo12:LC2
    b       puts
.L5:
; вывести "3, 4, 5"
    adrp    x0, .LC1
    add     x0, x0, :lo12:LC1
    b       puts
.L3:
; вывести "1, 2, 7, 10"
    adrp    x0, .LC0
    add     x0, x0, :lo12:LC0
    b       puts
.LC0:
    .string "1, 2, 7, 10"
.LC1:
    .string "3, 4, 5"
.LC2:
    .string "8, 9, 21"
.LC3:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
.string "22"
.LC4:
.string "default"
```

Скомпилируем этот пример как объектный файл и откроем его в **IDA!**. Вот таблица переходов:

Листинг 1.166: jumtable in IDA

```
.rodata:0000000000000064      AREA .rodata, DATA, READONLY
.rodata:0000000000000064      ; ORG 0x64
.rodata:0000000000000064 $d   DCB    9    ; case 1
.rodata:0000000000000065      DCB    9    ; case 2
.rodata:0000000000000066      DCB    6    ; case 3
.rodata:0000000000000067      DCB    6    ; case 4
.rodata:0000000000000068      DCB    6    ; case 5
.rodata:0000000000000069      DCB    6    ; case 6
.rodata:000000000000006A      DCB    9    ; case 7
.rodata:000000000000006B      DCB    3    ; case 8
.rodata:000000000000006C      DCB    3    ; case 9
.rodata:000000000000006D      DCB    9    ; case 10
.rodata:000000000000006E      DCB 0xF7  ; case 11
.rodata:000000000000006F      DCB 0xF7  ; case 12
.rodata:0000000000000070      DCB 0xF7  ; case 13
.rodata:0000000000000071      DCB 0xF7  ; case 14
.rodata:0000000000000072      DCB 0xF7  ; case 15
.rodata:0000000000000073      DCB 0xF7  ; case 16
.rodata:0000000000000074      DCB 0xF7  ; case 17
.rodata:0000000000000075      DCB 0xF7  ; case 18
.rodata:0000000000000076      DCB 0xF7  ; case 19
.rodata:0000000000000077      DCB    3    ; case 20
.rodata:0000000000000078      DCB    3    ; case 21
.rodata:0000000000000079      DCB    0    ; case 22
.rodata:000000000000007B ; .rodata ends
```

В случае 1, 9 будет умножено на 9 и прибавлено к адресу метки Lrtx4.

В случае 22, 0 будет умножено на 4, в результате это 0.

Место сразу за меткой Lrtx4 это метка L7, где находится код, выводящий «22».

В сегменте кода нет таблицы переходов, место для нее выделено в отдельной секции .rodata (нет особой нужды располагать её в сегменте кода).

Там есть также отрицательные байты (0xF7). Они используются для перехода назад, на код, выводящий строку «default» (на .L2).

#### 1.21.4. Fall-through

Ещё одно популярное использование оператора `switch()` это т.н. «fallthrough» («провал»). Вот простой пример<sup>98</sup>:

<sup>98</sup>Взято отсюда: [https://github.com/azonalon/prgraas/blob/master/proglib/lecture\\_examples/is\\_whitespace.c](https://github.com/azonalon/prgraas/blob/master/proglib/lecture_examples/is_whitespace.c)

```

1 bool is_whitespace(char c) {
2     switch (c) {
3         case ' ': // fallthrough
4         case '\t': // fallthrough
5         case '\r': // fallthrough
6         case '\n':
7             return true;
8         default: // not whitespace
9             return false;
10    }
11 }

```

Немного сложнее, из ядра Linux<sup>99</sup>:

```

1 char nco1, nco2;
2
3 void f(int if_freq_khz)
4 {
5
6     switch (if_freq_khz) {
7         default:
8             printf("IF=%d KHz is not supported, 3250 assumed\n",
9                 if_freq_khz);
10            /* fallthrough */
11            case 3250: /* 3.25Mhz */
12                nco1 = 0x34;
13                nco2 = 0x00;
14                break;
15            case 3500: /* 3.50Mhz */
16                nco1 = 0x38;
17                nco2 = 0x00;
18                break;
19            case 4000: /* 4.00Mhz */
20                nco1 = 0x40;
21                nco2 = 0x00;
22                break;
23            case 5000: /* 5.00Mhz */
24                nco1 = 0x50;
25                nco2 = 0x00;
26                break;
27            case 5380: /* 5.38Mhz */
28                nco1 = 0x56;
29                nco2 = 0x14;
30                break;
31    };

```

#### Листинг 1.167: Оптимизирующий GCC 5.4.0 x86

```
1 .LC0:
```

<sup>99</sup><https://github.com/torvalds/linux/blob/master/drivers/media/dvb-frontends/lgdt3306a.c>

```

2      .string "IF=%d KHz is not supported, 3250 assumed\n"
3  f:
4      sub     esp, 12
5      mov     eax, DWORD PTR [esp+16]
6      cmp     eax, 4000
7      je     .L3
8      jg     .L4
9      cmp     eax, 3250
10     je     .L5
11     cmp     eax, 3500
12     jne    .L2
13     mov     BYTE PTR nco1, 56
14     mov     BYTE PTR nco2, 0
15     add     esp, 12
16     ret
17  .L4:
18     cmp     eax, 5000
19     je     .L7
20     cmp     eax, 5380
21     jne    .L2
22     mov     BYTE PTR nco1, 86
23     mov     BYTE PTR nco2, 20
24     add     esp, 12
25     ret
26  .L2:
27     sub     esp, 8
28     push    eax
29     push    OFFSET FLAT:.LC0
30     call   printf
31     add     esp, 16
32  .L5:
33     mov     BYTE PTR nco1, 52
34     mov     BYTE PTR nco2, 0
35     add     esp, 12
36     ret
37  .L3:
38     mov     BYTE PTR nco1, 64
39     mov     BYTE PTR nco2, 0
40     add     esp, 12
41     ret
42  .L7:
43     mov     BYTE PTR nco1, 80
44     mov     BYTE PTR nco2, 0
45     add     esp, 12
46     ret

```

На метку `.L5` управление может перейти если на входе ф-ции число 3250. Но на эту метку можно попасть и с другой стороны: мы видим что между вызовом `printf()` и меткой `.L5` нет никаких переходов.

Теперь мы можем понять, почему иногда `switch()` является источником ошибок: если забыть дописать `break`, это прекратит выражение `switch()` в `fallthrough`, и вместо одного блока для каждого условия, будет исполняться сразу несколь-

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

ко.

## 1.21.5. Упражнения

### Упражнение#1

Вполне возможно переделать пример на Си в листинге ?? (стр. ??) так, чтобы при компиляции получалось даже ещё меньше кода, но работать всё будет точно так же. Попробуйте этого добиться.

## 1.22. Циклы

### 1.22.1. Простой пример

#### x86

Для организации циклов в архитектуре x86 есть старая инструкция L00P. Она проверяет значение регистра ECX и если оно не 0, делает [декремент](#) ECX и переход по метке, указанной в операнде. Возможно, эта инструкция не слишком удобная, потому что уже почти не бывает современных компиляторов, которые использовали бы её. Так что если вы видите где-то L00P, то с большой вероятностью это вручную написанный код на ассемблере.

Обычно, циклы на Си/Си++ создаются при помощи `for()`, `while()`, `do/while()`. Начнем с `for()`. Это выражение описывает инициализацию, условие, операцию после каждой итерации ([инкремент/декремент](#)) и тело цикла.

```
for (инициализация; условие; после каждой итерации)
{
    тело_цикла;
}
```

Примерно так же, генерируемый код и будет состоять из этих четырех частей. Возьмем пример:

```
#include <stdio.h>

void printing_function(int i)
{
    printf ("f(%d)\n", i);
};

int main()
{
    int i;

    for (i=2; i<10; i++)
        printing_function(i);

    return 0;
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

};

Имеем в итоге (MSVC 2010):

Листинг 1.168: MSVC 2010

```

_i$ = -4
_main PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     DWORD PTR _i$[ebp], 2 ; инициализация цикла
    jmp     SHORT $LN3@main
$LN2@main:
    mov     eax, DWORD PTR _i$[ebp] ; то что мы делаем после каждой итерации:
    add     eax, 1 ; добавляем 1 к i
    mov     DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp     DWORD PTR _i$[ebp], 10 ; это условие проверяется перед каждой
    итерацией
    jge     SHORT $LN1@main ; если i больше или равно 10, заканчиваем
    цикл
    mov     ecx, DWORD PTR _i$[ebp] ; тело цикла: вызов функции
    printing_function(i)
    push    ecx
    call    _printing_function
    add     esp, 4
    jmp     SHORT $LN2@main ; переход на начало цикла
$LN1@main:
    ; конец цикла
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main ENDP

```

В принципе, ничего необычного.

GCC 4.4.1 выдает примерно такой же код, с небольшой разницей:

Листинг 1.169: GCC 4.4.1

```

main      proc near

var_20    = dword ptr -20h
var_4     = dword ptr -4

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFF0h
        sub     esp, 20h
        mov     [esp+20h+var_4], 2 ; инициализация i
        jmp     short loc_8048476

loc_8048465:
        mov     eax, [esp+20h+var_4]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        mov     [esp+20h+var_20], eax
        call   printing_function
        add     [esp+20h+var_4], 1 ; инкремент i

loc_8048476:
        cmp     [esp+20h+var_4], 9
        jle    short loc_8048465 ; если i<=9, продолжаем цикл
        mov     eax, 0
        leave
        retn
main     endp

```

Интересно становится, если скомпилируем этот же код при помощи MSVC 2010 с включенной оптимизацией (/Ox):

Листинг 1.170: Оптимизирующий MSVC

```

_main   PROC
        push   esi
        mov    esi, 2
$LL3@main:
        push   esi
        call  _printing_function
        inc    esi
        add    esp, 4
        cmp    esi, 10 ; 0000000aH
        jl    SHORT $LL3@main
        xor    eax, eax
        pop    esi
        ret    0
_main   ENDP

```

Здесь происходит следующее: переменную *i* компилятор не выделяет в локальном стеке, а выделяет целый регистр под нее: ESI. Это возможно для маленьких функций, где мало локальных переменных.

В принципе, всё то же самое, только теперь одна важная особенность: *f()* не должна менять значение ESI. Наш компилятор уверен в этом, а если бы и была необходимость использовать регистр ESI в функции *f()*, то её значение сохранялось бы в стеке. Примерно так же как и в нашем листинге: обратите внимание на PUSH ESI/POP ESI в начале и конце функции.

Попробуем GCC 4.4.1 с максимальной оптимизацией (-O3):

Листинг 1.171: Оптимизирующий GCC 4.4.1

```

main     proc near

var_10   = dword ptr -10h

        push   ebp
        mov    ebp, esp
        and    esp, 0FFFFFF0h
        sub    esp, 10h

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov     [esp+10h+var_10], 2
call   printing_function
mov     [esp+10h+var_10], 3
call   printing_function
mov     [esp+10h+var_10], 4
call   printing_function
mov     [esp+10h+var_10], 5
call   printing_function
mov     [esp+10h+var_10], 6
call   printing_function
mov     [esp+10h+var_10], 7
call   printing_function
mov     [esp+10h+var_10], 8
call   printing_function
mov     [esp+10h+var_10], 9
call   printing_function
xor     eax, eax
leave
retn
main    endp

```

Однако GCC просто *развернул* цикл<sup>100</sup>.

Делается это в тех случаях, когда итераций не слишком много (как в нашем примере) и можно немного сэкономить время, убрав все инструкции, обеспечивающие цикл. В качестве обратной стороны медали, размер кода увеличился.

Использовать большие развернутые циклы в наше время не рекомендуется, потому что большие функции требуют больше кэш-памяти<sup>101</sup>.

Увеличим максимальное значение  $i$  в цикле до 100 и попробуем снова. GCC выдает:

Листинг 1.172: GCC

```

main    public main
        proc near
var_20  = dword ptr -20h

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFF0h
        push    ebx
        mov     ebx, 2    ; i=2
        sub     esp, 1Ch

; выравнивание метки loc_80484D0 (начало тела цикла)
; по 16-байтной границе:
        nop

```

<sup>100</sup>[loop unwinding](#) в англоязычной литературе

<sup>101</sup>Очень хорошая статья об этом: [Ulrich Drepper, *What Every Programmer Should Know About Memory*, (2007)]<sup>102</sup>. А также о рекомендациях о развернутых циклах от Intel можно прочитать здесь: [Intel® 64 and IA-32 Architectures Optimization Reference Manual, (2014)3.4.1.7].

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```
loc_80484D0:
; передать i как первый аргумент для printing_function():
    mov     [esp+20h+var_20], ebx
    add     ebx, 1      ; i++
    call   printing_function
    cmp     ebx, 64h   ; i==100?
    jnz     short loc_80484D0 ; если нет, продолжать
    add     esp, 1Ch
    xor     eax, eax   ; возврат 0
    pop     ebx
    mov     esp, ebp
    pop     ebp
    retn
main      endp
```

Это уже похоже на то, что сделал MSVC 2010 в режиме оптимизации (/Ox). За исключением того, что под переменную *i* будет выделен регистр EBX.

GCC уверен, что этот регистр не будет модифицироваться внутри *f()*, а если вдруг это и придётся там сделать, то его значение будет сохранено в начале функции, прямо как в *main()*.

## x86: OllyDbg

Скомпилируем наш пример в MSVC 2010 с /Ox и /Ob0 и загрузим в OllyDbg.

Оказывается, OllyDbg может обнаруживать простые циклы и показывать их в квадратных скобках, для удобства:

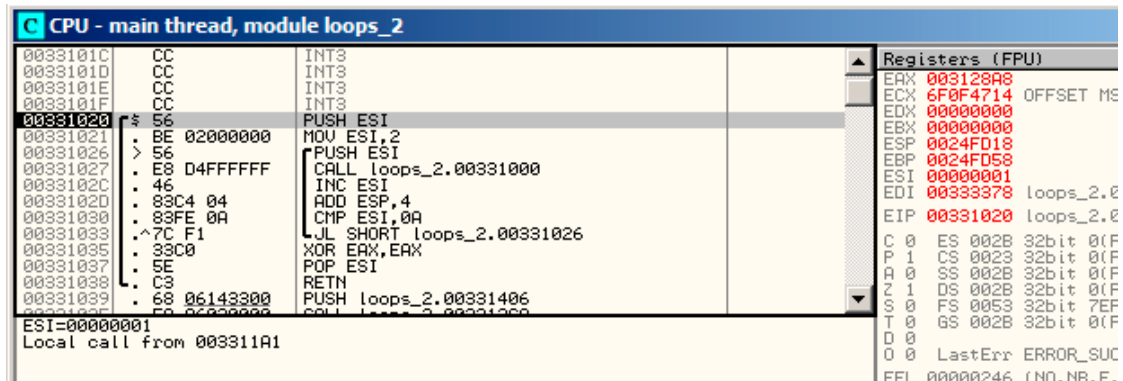


Рис. 1.54: OllyDbg: начало main()

Трассируя (F8 — сделать шаг, не входя в функцию) мы видим, как ESI увеличивается на 1.

Например, здесь  $ESI = i = 6$ :

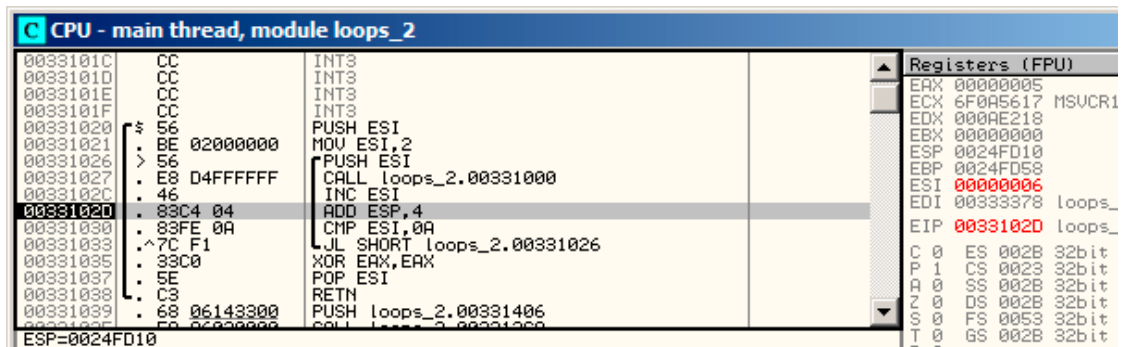
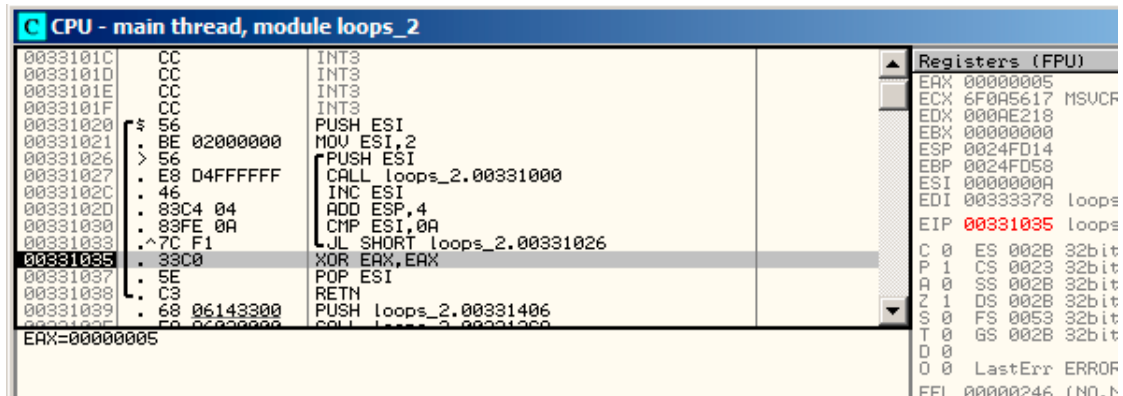


Рис. 1.55: OllyDbg: тело цикла только что отработало с  $i = 6$

9 это последнее значение цикла. Поэтому JL после **инкремента** не срабатывает и функция заканчивается:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Рис. 1.56: OllyDbg: *ESI* = 10, конец цикла**x86: tracer**

Как видно, трассировать вручную цикл в отладчике — это не очень удобно. Поэтому попробуем **tracer**. Открываем скомпилированный пример в **IDA!**, находим там адрес инструкции PUSH ESI (передающей единственный аргумент в `f()`), а это 0x401026 в нашем случае и запускаем **tracer**:

```
tracer.exe -l:loops_2.exe bpx=loops_2.exe!0x00401026
```

Опция BPX просто ставит точку останова по адресу и затем tracer будет выдавать состояние регистров. В `tracer.log` после запуска я вижу следующее:

```
PID=12884|New process loops_2.exe
(0) loops_2.exe!0x401026
EAX=0x00a328c8 EBX=0x00000000 ECX=0x6f0f4714 EDX=0x00000000
ESI=0x00000002 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=PF ZF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000003 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000004 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000005 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

ESI=0x00000006 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000007 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000008 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000009 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
PID=12884|Process loops_2.exe exited. ExitCode=0 (0x0)

```

Видно, как значение ESI последовательно изменяется от 2 до 9. И даже более того, в **tracer** можно собирать значения регистров по всем адресам внутри функции.

Там это называется *trace*. Каждая инструкция трассируется, значения самых интересных регистров запоминаются. Затем генерируется .idc-скрипт для **IDA!**, который добавляет комментарии. Итак, в **IDA!** я узнал что адрес main() это 0x00401020 и запускаю:

```
tracer.exe -l:loops_2.exe bpf=loops_2.exe!0x00401020,trace:cc
```

BPF означает установить точку останова на функции.

Получаю в итоге скрипты loops\_2.exe.idc и loops\_2.exe\_clear.idc.

Загружаю loops\_2.exe.idc в **IDA!** и увижу следующее:

```
.text:00401020
.text:00401020 ; ===== SUBROUTINE =====
.text:00401020
.text:00401020 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401020 _main      proc near          ; CODE XREF: __tmainCRTStartup+11D.jp
.text:00401020          = dword ptr 4
.text:00401020 argv      = dword ptr 8
.text:00401020 envp     = dword ptr 0Ch
.text:00401020          push     esi          ; ESI=1
.text:00401021          mov     esi, 2
.text:00401026 loc_401026:          ; CODE XREF: _main+13.jj
.text:00401026          push     esi          ; ESI=2..9
.text:00401027          call    sub_401000    ; tracing nested maximum level (1) reached,
.text:0040102c          inc     esi           ; ESI=2..9
.text:0040102d          add     esp, 4        ; ESP=0x38fcbc
.text:00401030          cmp     esi, 0Ah     ; ESI=3..0xa
.text:00401033          jl     short loc_401026 ; SF=false,true OF=false
.text:00401035          xor     eax, eax
.text:00401037          pop     esi
.text:00401038          retn                    ; EAX=0
.text:00401038 _main      endp
```

Рис. 1.57: **IDA!** с загруженным .idc-скриптом

Видно, что ESI меняется от 2 до 9 в начале тела цикла, но после **инкремента** он в пределах [3..0xA]. Видно также, что функция main() заканчивается с 0 в EAX.

**tracer** также генерирует loops\_2.exe.txt, содержащий адреса инструкций, сколько раз была исполнена каждая и значения регистров:

Листинг 1.173: loops\_2.exe.txt

0x401020 (.text+0x20), e=	1 [PUSH ESI] ESI=1
0x401021 (.text+0x21), e=	1 [MOV ESI, 2]
0x401026 (.text+0x26), e=	8 [PUSH ESI] ESI=2..9
0x401027 (.text+0x27), e=	8 [CALL 8D1000h] tracing nested maximum ↗ ↳ level (1) reached, skipping this CALL 8D1000h=0x8d1000
0x40102c (.text+0x2c), e=	8 [INC ESI] ESI=2..9
0x40102d (.text+0x2d), e=	8 [ADD ESP, 4] ESP=0x38fcbc
0x401030 (.text+0x30), e=	8 [CMP ESI, 0Ah] ESI=3..0xa
0x401033 (.text+0x33), e=	8 [JL 8D1026h] SF=false,true OF=false
0x401035 (.text+0x35), e=	1 [XOR EAX, EAX]
0x401037 (.text+0x37), e=	1 [POP ESI]
0x401038 (.text+0x38), e=	1 [RETN] EAX=0

Так можно использовать **grep**.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

**ARM****Неоптимизирующий Keil 6/2013 (Режим ARM)**

```

main
    STMFD    SP!, {R4,LR}
    MOV     R4, #2
    B       loc_368
loc_35C    ; CODE XREF: main+1C
    MOV     R0, R4
    BL     printing_function
    ADD     R4, R4, #1

loc_368    ; CODE XREF: main+8
    CMP     R4, #0xA
    BLT     loc_35C
    MOV     R0, #0
    LDMFD   SP!, {R4,PC}

```

Счетчик итераций  $i$  будет храниться в регистре R4. Инструкция `MOV R4, #2` просто инициализирует  $i$ . Инструкции `MOV R0, R4` и `BL printing_function` составляют тело цикла. Первая инструкция готовит аргумент для функции, `f()` а вторая вызывает её. Инструкция `ADD R4, R4, #1` прибавляет единицу к  $i$  при каждой итерации. `CMP R4, #0xA` сравнивает  $i$  с `0xA` (10). Следующая за ней инструкция `BLT (Branch Less Than)` совершит переход, если  $i$  меньше чем 10. В противном случае в R0 запишется 0 (потому что наша функция возвращает 0) и произойдет выход из функции.

**Оптимизирующий Keil 6/2013 (Режим Thumb)**

```

_main
    PUSH    {R4,LR}
    MOVS   R4, #2

loc_132    ; CODE XREF: _main+E
    MOVS   R0, R4
    BL     printing_function
    ADDS   R4, R4, #1
    CMP    R4, #0xA
    BLT    loc_132
    MOVS   R0, #0
    POP    {R4,PC}

```

Практически всё то же самое.

**Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)**

```

_main
    PUSH    {R4,R7,LR}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

MOVW      R4, #0x1124 ; "%d\n"
MOVS      R1, #2
MOVT.W    R4, #0
ADD       R7, SP, #4
ADD       R4, PC
MOV       R0, R4
BLX      _printf
MOV       R0, R4
MOVS      R1, #3
BLX      _printf
MOV       R0, R4
MOVS      R1, #4
BLX      _printf
MOV       R0, R4
MOVS      R1, #5
BLX      _printf
MOV       R0, R4
MOVS      R1, #6
BLX      _printf
MOV       R0, R4
MOVS      R1, #7
BLX      _printf
MOV       R0, R4
MOVS      R1, #8
BLX      _printf
MOV       R0, R4
MOVS      R1, #9
BLX      _printf
MOVS      R0, #0
POP       {R4,R7,PC}

```

На самом деле, в моей функции `f()` было такое:

```

void printing_function(int i)
{
    printf ("%d\n", i);
};

```

Так что LLVM не только *развернул* цикл, но также и представил мою очень простую функцию `f()` как *inline-функцию*, и вставил её тело вместо цикла 8 раз. Это возможно, когда функция очень простая (как та что у меня) и когда она вызывается не очень много раз, как здесь.

## ARM64: Оптимизирующий GCC 4.9.1

Листинг 1.174: Оптимизирующий GCC 4.9.1

```

printing_function:
; подготовить второй аргумент printf():
    mov     w1, w0
; загрузить адрес строки "f(%d)\n"
    adrp   x0, .LC0

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        add    x0, x0, :lo12:LC0
; здесь просто переход вместо перехода с сохранением адреса и инструкции
; возврата:
        b     printf
main:
; сохранить FP и LR в локальном стеке:
        stp   x29, x30, [sp, -32]!
; установить стековый фрейм:
        add   x29, sp, 0
; сохранить содержимое регистра X19 в локальном стеке:
        str   x19, [sp,16]
; будем использовать регистр W19 как счетчик.
; установить начальное значение в 2:
        mov   w19, 2
.L3:
; подготовить первый аргумент printing_function():
        mov   w0, w19
; инкремент регистра счетчика.
        add   w19, w19, 1
; W0 все еще содержит значение счетчика перед инкрементом.
        bl   printing_function
; конец?
        cmp   w19, 10
; нет, перейти на начало тела цикла:
        bne   .L3
; возврат 0
        mov   w0, 0
; восстановить содержимое регистра X19:
        ldr   x19, [sp,16]
; восстановить значения FP и LR:
        ldp   x29, x30, [sp], 32
        ret
.LC0:
        .string "f(%d)\n"

```

## ARM64: Неоптимизирующий GCC 4.9.1

Листинг 1.175: Неоптимизирующий GCC 4.9.1 -fno-inline

```

.LC0:
        .string "f(%d)\n"
printing_function:
; сохранить FP и LR в локальном стеке:
        stp   x29, x30, [sp, -32]!
; установить стековый фрейм:
        add   x29, sp, 0
; сохранить содержимое регистра W0:
        str   w0, [x29,28]
; загрузить адрес строки "f(%d)\n"
        adrp  x0, .LC0
        add   x0, x0, :lo12:LC0
; перезагрузить входное значение из локального стека в регистр W1:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

    ldr    w1, [x29,28]
; вызвать printf()
    bl    printf
; восстановить значения FP и LR:
    ldp   x29, x30, [sp], 32
    ret
main:
; сохранить FP и LR в локальном стеке:
    stp   x29, x30, [sp, -32]!
; установить стековый фрейм:
    add   x29, sp, 0
; инициализировать счетчик
    mov   w0, 2
; сохранить его в месте, выделенном для него в локальном стеке:
    str   w0, [x29,28]
; пропустить тело цикла, и перейти на инструкции проверки условия цикла:
    b     .L3
.L4:
; загрузить значение счетчика в W0.
; это будет первый аргумент ф-ции printing_function():
    ldr   w0, [x29,28]
; вызвать printing_function():
    bl   printing_function
; инкремент значения счетчика:
    ldr   w0, [x29,28]
    add   w0, w0, 1
    str   w0, [x29,28]
.L3:
; проверка условия цикла.
; загрузить значение счетчика:
    ldr   w0, [x29,28]
; это 9?
    cmp   w0, 9
; меньше или равно? тогда перейти на начало тела цикла:
; иначе ничего не делаем.
    ble   .L4
; возврат 0
    mov   w0, 0
; восстановить значения FP и LR:
    ldp   x29, x30, [sp], 32
    ret

```

## MIPS

Листинг 1.176: Неоптимизирующий GCC 4.4.5 (IDA)

```

main:
; IDA не знает названия переменных в локальном стеке
; Это мы назвали их вручную:
i      = -0x10
saved_FP = -8
saved_RA = -4

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; пролог функции:
    addiu    $sp, -0x28
    sw      $ra, 0x28+saved_RA($sp)
    sw      $fp, 0x28+saved_FP($sp)
    move    $fp, $sp
; инициализировать счетчик значением 2 и сохранить это значение в локальном
стеке
    li      $v0, 2
    sw      $v0, 0x28+i($fp)
; псевдоинструкция. здесь на самом деле "BEQ $ZERO, $ZERO, loc_9C":
    b       loc_9C
    or      $at, $zero ; branch delay slot, NOP

loc_80:                                # CODE XREF: main+48
; загрузить значение счетчика из локального стека и вызвать
printing_function():
    lw      $a0, 0x28+i($fp)
    jal    printing_function
    or      $at, $zero ; branch delay slot, NOP
; загрузить счетчик, инкрементировать его и записать назад:
    lw      $v0, 0x28+i($fp)
    or      $at, $zero ; NOP
    addiu   $v0, 1
    sw      $v0, 0x28+i($fp)

loc_9C:                                # CODE XREF: main+18
; проверить счетчик, он больше 10?
    lw      $v0, 0x28+i($fp)
    or      $at, $zero ; NOP
    slti   $v0, 0xA
; если он меньше 10, перейти на loc_80 (начало тела цикла):
    bnez   $v0, loc_80
    or      $at, $zero ; branch delay slot, NOP
; заканчиваем, возвращаем 0:
    move    $v0, $zero
; эпилог функции:
    move    $sp, $fp
    lw      $ra, 0x28+saved_RA($sp)
    lw      $fp, 0x28+saved_FP($sp)
    addiu   $sp, 0x28
    jr     $ra
    or      $at, $zero ; branch delay slot, NOP

```

Новая для нас инструкция это B. Вернее, это псевдоинструкция (BEQ).

### Ещё кое-что

По генерируемому коду мы видим следующее: после инициализации *i*, тело цикла не исполняется. В начале проверяется условие *i*, а лишь затем исполняется тело цикла. И это правильно, потому что если условие в самом начале не истинно, тело цикла исполнять нельзя.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Так может быть, например, в таком случае:

```
for (i=0; i<количество_элементов_для_обработки; i++)
    тело_цикла;
```

Если *количество\_элементов\_для\_обработки* равно 0, тело цикла не должно исполниться ни разу. Поэтому проверка условия происходит перед тем как исполнить само тело.

Впрочем, оптимизирующий компилятор может переставить проверку условия и тело цикла местами, если он уверен, что описанная здесь ситуация невозможна, как в случае с нашим простейшим примером и с применением компиляторов Keil, Xcode (LLVM), MSVC и GCC в режиме оптимизации.

### 1.22.2. Функция копирования блоков памяти

Настоящие функции копирования памяти могут копировать по 4 или 8 байт на каждой итерации, использовать **SIMD!**<sup>103</sup>, векторизацию, и т. д.

Но ради простоты, этот пример настолько прост, насколько это возможно.

```
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
    size_t i;
    for (i=0; i<cnt; i++)
        dst[i]=src[i];
};
```

### Простейшая реализация

Листинг 1.177: GCC 4.9 x64 оптимизация по размеру (-Os)

```
my_memcpy:
; RDI = целевой адрес
; RSI = исходный адрес
; RDX = размер блока

; инициализировать счетчик (i) в 0
xor     eax, eax
.L2:
; все байты скопированы? тогда заканчиваем:
cmp     rax, rdx
je     .L5
; загружаем байт по адресу RSI+i:
mov     cl, BYTE PTR [rsi+rax]
; записываем байт по адресу RDI+i:
mov     BYTE PTR [rdi+rax], cl
inc     rax ; i++
jmp     .L2
```

<sup>103</sup>**SIMD!**

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
.L5:
    ret
```

Листинг 1.178: GCC 4.9 ARM64 оптимизация по размеру (-Os)

```
my_memcpy:
; X0 = целевой адрес
; X1 = исходный адрес
; X2 = размер блока

; инициализировать счетчик (i) в 0
    mov    x3, 0
.L2:
; все байты скопированы? тогда заканчиваем:
    cmp    x3, x2
    beq    .L5
; загружаем байт по адресу X1+i:
    ldrb   w4, [x1,x3]
; записываем байт по адресу X0+i:
    strb   w4, [x0,x3]
    add    x3, x3, 1 ; i++
    b     .L2
.L5:
    ret
```

Листинг 1.179: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
my_memcpy PROC
; R0 = целевой адрес
; R1 = исходный адрес
; R2 = размер блока

    PUSH   {r4,lr}
; инициализировать счетчик (i) в 0
    MOVS   r3,#0
; условие проверяется в конце ф-ции, так что переходим туда:
    B      |L0.12|
|L0.6|
; загружаем байт по адресу R1+i:
    LDRB   r4,[r1,r3]
; записываем байт по адресу R0+i:
    STRB   r4,[r0,r3]
; i++
    ADDS   r3,r3,#1
|L0.12|
; i<size?
    CMP    r3,r2
; перейти на начало цикла, если это так:
    BCC   |L0.6|
    POP   {r4,pc}
    ENDP
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мой адреса](#). Спасибо!

**ARM в режиме ARM**

Keil в режиме ARM пользуется условными суффиксами:

Листинг 1.180: Оптимизирующий Keil 6/2013 (Режим ARM)

```

my_memcpy PROC
; R0 = целевой адрес
; R1 = исходный адрес
; R2 = размер блока

; инициализировать счетчик (i) в 0
    MOV     r3,#0
|L0.4|
; все байты скопированы?
    CMP     r3,r2
; следующий блок исполнится только в случае условия меньше чем,
; т.е., если R2<R3 или i<size.
; загружаем байт по адресу R1+i:
    LDRBCC  r12,[r1,r3]
; записываем байт по адресу R0+i:
    STRBCC  r12,[r0,r3]
; i++
    ADDCC   r3,r3,#1
; последняя инструкция условного блока.
; перейти на начало цикла, если i<size
; в противном случае, ничего не делать (т.е. если i>=size)
    BCC     |L0.4|
; возврат
    BX     lr
    ENDP

```

Вот почему здесь только одна инструкция перехода вместо двух.

**MIPS**

Листинг 1.181: GCC 4.4.5 оптимизация по размеру (-Os) (IDA)

```

my_memcpy:
; перейти на ту часть цикла, где проверяется условие:
    b      loc_14
; инициализировать счетчик (i) в 0
; он будет всегда находиться в регистре $v0:
    move   $v0, $zero ; branch delay slot

loc_8:                                     # CODE XREF: my_memcpy+1C
; загрузить байт как беззнаковый по адресу $t0 в $v1:
    lbu   $v1, 0($t0)
; инкремент счетчика (i):
    addiu $v0, 1
; записываем байт по адресу $a3
    sb    $v1, 0($a3)

loc_14:                                     # CODE XREF: my_memcpy

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; проверить, до сих пор ли счетчик (i) в $v0 меньше чем третий аргумент
  ("cnt" в $a2)
    sltu    $v1, $v0, $a2
; сформировать адрес байта исходного блока:
    addu   $t0, $a1, $v0
; $t0 = $a1+$v0 = src+i
; перейти на тело цикла, если счетчик всё еще меньше чем "cnt":
    bnez   $v1, loc_8
; сформировать адрес байта в целевом блоке ($a3 = $a0+$v0 = dst+i):
    addu   $a3, $a0, $v0 ; branch delay slot
; закончить, если BNEZ не сработала
    jr     $ra
    or     $at, $zero ; branch delay slot, NOP

```

Здесь две новых для нас инструкций: LBU («Load Byte Unsigned») и SB («Store Byte»). Так же как и в ARM, все регистры в MIPS имеют длину в 32 бита. Здесь нет частей регистров равных байту, как в x86.

Так что когда нужно работать с байтами, приходится выделять целый 32-битный регистр для этого.

LBU загружает байт и сбрасывает все остальные биты («Unsigned»).

И напротив, инструкция LB («Load Byte») расширяет байт до 32-битного значения учитывая знак.

SB просто записывает байт из младших 8 бит регистра в память.

## Векторизация

Оптимизирующий GCC может из этого примера сделать намного больше: ?? (стр. ??).

### 1.22.3. Проверка условия

Важно помнить, что в конструкции *for()*, проверка условия происходит не в конце, а в начале, перед исполнением тела цикла. Но нередко компилятору удобнее проверять условие в конце, после тела. Иногда может добавляться еще одна проверка в начале.

Например:

```

#include <stdio.h>

void f(int start, int finish)
{
    for (; start<finish; start++)
        printf ("%d\n", start);
};

```

Оптимизирующий GCC 5.4.0 x64:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
f:
; check condition (1):
    cmp     edi, esi
    jge     .L9
    push   rbp
    push   rbx
    mov    ebp, esi
    mov    ebx, edi
    sub    rsp, 8
.L5:
    mov    edx, ebx
    xor    eax, eax
    mov    esi, OFFSET FLAT:.LC0 ; "%d\n"
    mov    edi, 1
    add    ebx, 1
    call   __printf_chk
; check condition (2):
    cmp    ebp, ebx
    jne    .L5
    add    rsp, 8
    pop    rbx
    pop    rbp
.L9:
    rep ret
```

Видим две проверки.

Hex-Rays (по крайней мере версии 2.2.0) декомпилирует это так:

```
void __cdecl f(unsigned int start, unsigned int finish)
{
    unsigned int v2; // ebx@2
    __int64 v3; // rdx@3

    if ( (signed int)start < (signed int)finish )
    {
        v2 = start;
        do
        {
            v3 = v2++;
            _printf_chk(1LL, "%d\n", v3);
        }
        while ( finish != v2 );
    }
}
```

В данном случае, *do/while()* можно смело заменять на *for()*, а первую проверку убрать.

#### 1.22.4. Вывод

Примерный скелет цикла от 2 до 9 включительно:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Листинг 1.182: x86

```

mov [counter], 2 ; инициализация
jmp check
body:
; тело цикла
; делаем тут что-нибудь
; используем переменную счетчика в локальном стеке
add [counter], 1 ; инкремент
check:
cmp [counter], 9
jle body

```

Операция инкремента может быть представлена как 3 инструкции в неоптимизированном коде:

Листинг 1.183: x86

```

MOV [counter], 2 ; инициализация
JMP check
body:
; тело цикла
; делаем тут что-нибудь
; используем переменную счетчика в локальном стеке
MOV REG, [counter] ; инкремент
INC REG
MOV [counter], REG
check:
CMP [counter], 9
JLE body

```

Если тело цикла короткое, под переменную счетчика можно выделить целый регистр:

Листинг 1.184: x86

```

MOV EBX, 2 ; инициализация
JMP check
body:
; тело цикла
; делаем тут что-нибудь
; используем переменную счетчика в EBX, но не изменяем её!
INC EBX ; инкремент
check:
CMP EBX, 9
JLE body

```

Некоторые части цикла могут быть сгенерированы компилятором в другом порядке:

Листинг 1.185: x86

```

MOV [counter], 2 ; инициализация
JMP label_check
label_increment:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

    ADD [counter], 1 ; инкремент
label_check:
    CMP [counter], 10
    JGE exit
    ; тело цикла
    ; делаем тут что-нибудь
    ; используем переменную счетчика в локальном стеке
    JMP label_increment
exit:

```

Обычно условие проверяется *перед* телом цикла, но компилятор может пере-  
строить цикл так, что условие проверяется *после* тела цикла.

Это происходит тогда, когда компилятор уверен, что условие всегда будет  
*истинно* на первой итерации, так что тело цикла исполнится как минимум один  
раз:

#### Листинг 1.186: x86

```

    MOV REG, 2 ; инициализация
body:
    ; тело цикла
    ; делаем тут что-нибудь
    ; используем переменную счетчика в REG, но не изменяем её!
    INC REG ; инкремент
    CMP REG, 10
    JL body

```

Используя инструкцию LOOP. Это редкость, компиляторы не используют её. Так  
что если вы её видите, это верный знак, что этот фрагмент кода написан вруч-  
ную:

#### Листинг 1.187: x86

```

    ; считать от 10 до 1
    MOV ECX, 10
body:
    ; тело цикла
    ; делаем тут что-нибудь
    ; используем переменную счетчика в ECX, но не изменяем её!
    LOOP body

```

ARM. В этом примере регистр R4 выделен для переменной счетчика:

#### Листинг 1.188: ARM

```

    MOV R4, 2 ; инициализация
    B check
body:
    ; тело цикла
    ; делаем тут что-нибудь
    ; используем переменную счетчика в R4, но не изменяем её!
    ADD R4,R4, #1 ; инкремент
check:

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения,  
пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
CMP R4, #10
BLT body
```

### 1.22.5. Упражнения

- <http://challenges.re/54>
- <http://challenges.re/55>
- <http://challenges.re/56>
- <http://challenges.re/57>

## 1.23. Ещё кое-что о строках

### 1.23.1. strlen()

Ещё немного о циклах. Часто функция `strlen()` <sup>104</sup> реализуется при помощи `while()`. Например, вот как это сделано в стандартных библиотеках MSVC:

```
int my_strlen (const char * str)
{
    const char *eos = str;

    while( *eos++ ) ;

    return( eos - str - 1 );
}

int main()
{
    // test
    return my_strlen("hello!");
};
```

## x86

### Неоптимизирующий MSVC

Итак, компилируем:

```
_eos$ = -4 ; size = 4
_str$ = 8 ; size = 4
_strlen PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _str$[ebp] ; взять указатель на символ из "str"
```

<sup>104</sup>подсчет длины строки в Си

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov     DWORD PTR _eos$[ebp], eax ; и переложить его в нашу локальную
      переменную "eos"
$LN2@strlen_:
mov     ecx, DWORD PTR _eos$[ebp] ; ECX=eos

; взять байт, на который указывает ECX и положить его
; в EDX расширяя до 32-х бит, учитывая знак

movsx   edx, BYTE PTR [ecx]
mov     eax, DWORD PTR _eos$[ebp] ; EAX=eos
add     eax, 1 ; инкремент EAX
mov     DWORD PTR _eos$[ebp], eax ; положить eax назад в "eos"
test    edx, edx ; EDX ноль?
je      SHORT $LN1@strlen_ ; да, то что лежит в EDX это ноль,
      выйти из цикла
jmp     SHORT $LN2@strlen_ ; продолжаем цикл
$LN1@strlen_:

; здесь мы вычисляем разницу двух указателей

mov     eax, DWORD PTR _eos$[ebp]
sub     eax, DWORD PTR _str$[ebp]
sub     eax, 1 ; отнимаем от разницы еще единицу и
      возвращаем результат
mov     esp, ebp
pop     ebp
ret     0
_strlen_ ENDP

```

Здесь две новых инструкции: MOVVSX и TEST.

О первой. MOVVSX предназначена для того, чтобы взять байт из какого-либо места в памяти и положить его, в нашем случае, в регистр EDX. Но регистр EDX — 32-битный. MOVVSX означает *MOV with Sign-Extend*. Оставшиеся биты с 8-го по 31-й MOVVSX сделает единицей, если исходный байт в памяти имеет знак *минус*, или заполнит нулями, если знак *плюс*.

И вот зачем всё это.

По умолчанию в MSVC и GCC тип *char* — знаковый. Если у нас есть две переменные, одна *char*, а другая *int* (*int* тоже знаковый), и если в первой переменной лежит -2 (что кодируется как 0xFE) и мы просто переложим это в *int*, то там будет 0x000000FE, а это, с точки зрения *int*, даже знакового, будет 254, но никак не -2. -2 в переменной *int* кодируется как 0xFFFFFFF2. Для того чтобы значение 0xFE из переменной типа *char* переложить в знаковый *int* с сохранением всего, нужно узнать его знак и затем заполнить остальные биты. Это делает MOVVSX.

Хотя конкретно здесь компилятору вряд ли была особая надобность хранить значение *char* в регистре EDX, а не его восьмибитной части, скажем DL. Но получилось, как получилось. Должно быть [register allocator](#) компилятора сработал именно так.

Позже выполняется TEST EDX, EDX. Об инструкции TEST читайте в разделе о битовых полях (?? (стр. ??)). Конкретно здесь эта инструкция просто проверяет

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

состояние регистра EDX на 0.

## Неоптимизирующий GCC

Попробуем GCC 4.4.1:

```

strlen      public strlen
strlen      proc near

eos         = dword ptr -4
arg_0      = dword ptr  8

            push    ebp
            mov     ebp, esp
            sub     esp, 10h
            mov     eax, [ebp+arg_0]
            mov     [ebp+eos], eax

loc_80483F0:
            mov     eax, [ebp+eos]
            movzx  eax, byte ptr [eax]
            test   al, al
            setnz  al
            add     [ebp+eos], 1
            test   al, al
            jnz   short loc_80483F0
            mov     edx, [ebp+eos]
            mov     eax, [ebp+arg_0]
            mov     ecx, edx
            sub     ecx, eax
            mov     eax, ecx
            sub     eax, 1
            leave
            retn
strlen      endp

```

Результат очень похож на MSVC, только здесь используется MOVZX, а не MOVZX. MOVZX означает *MOV with Zero-Extend*. Эта инструкция перекладывает какое-либо значение в регистр и остальные биты выставляет в 0. Фактически, преимущество этой инструкции только в том, что она позволяет заменить две инструкции сразу:

```
xor eax, eax / mov al, [...].
```

С другой стороны, нам очевидно, что здесь можно было бы написать вот так: `mov al, byte ptr [eax] / test al, al` — это тоже самое, хотя старшие биты EAX будут «замусорены». Но будем считать, что это погрешность компилятора — он не смог сделать код более экономным или более понятным. Строго говоря, компилятор вообще не нацелен на то, чтобы генерировать понятный (для человека) код.

Следующая новая инструкция для нас — SETNZ. В данном случае, если в AL был не ноль, то `test al, al` выставит флаг ZF в 0, а SETNZ, если ZF==0 (NZ значит

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

*not zero*) выставит 1 в AL. Смысл этой процедуры в том, что *если AL не ноль, выполнить переход на loc\_80483F0*. Компилятор выдал немного избыточный код, но не будем забывать, что оптимизация выключена.

### Оптимизирующий MSVC

Теперь скомпилируем всё то же самое в MSVC 2012, но с включенной оптимизацией (/Ox):

Листинг 1.189: Оптимизирующий MSVC 2012 /Ob0

```

_str$ = 8 ; size = 4
_strlen PROC
    mov     edx, DWORD PTR _str$[esp-4] ; EDX -> указатель на строку
    mov     eax, edx ; переложить в EAX
$LL2@strlen:
    mov     cl, BYTE PTR [eax] ; CL = *EAX
    inc     eax ; EAX++
    test    cl, cl ; CL==0?
    jne     SHORT $LL2@strlen ; нет, продолжаем цикл
    sub     eax, edx ; вычисляем разницу указателей
    dec     eax ; декремент EAX
    ret     0
_strlen ENDP

```

Здесь всё попроще стало. Но следует отметить, что компилятор обычно может так хорошо использовать регистры только на небольших функциях с небольшим количеством локальных переменных.

INC/DEC— это инструкции [инкремента-декремента](#). Попросту говоря — увеличить на единицу или уменьшить.

## Оптимизирующий MSVC + OllyDbg

Можем попробовать этот (соптимизированный) пример в OllyDbg. Вот самая первая итерация:

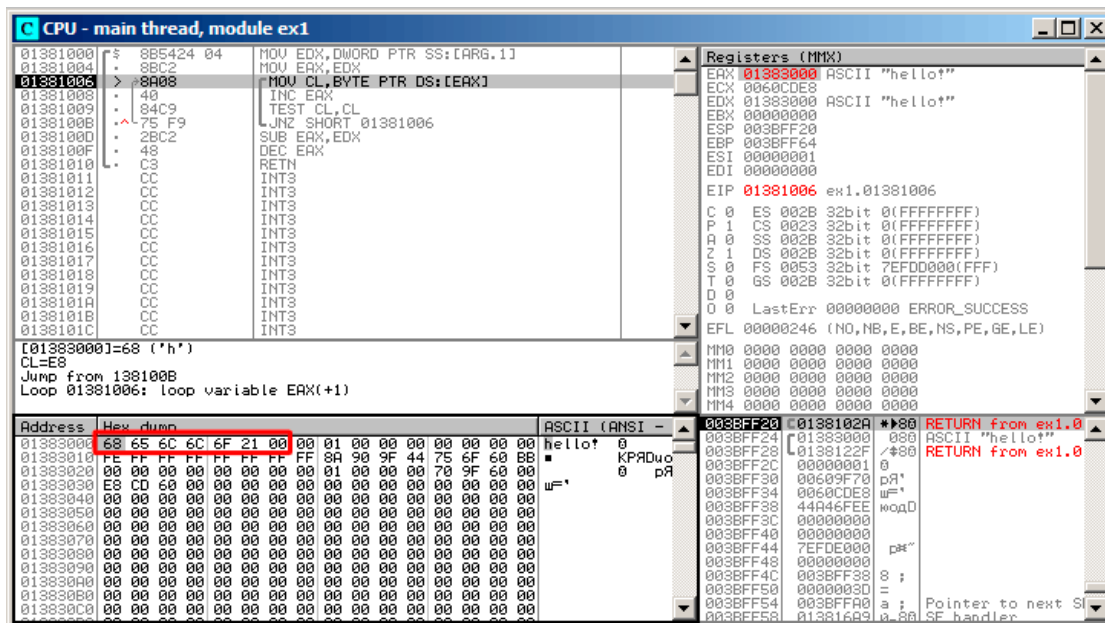


Рис. 1.58: OllyDbg: начало первой итерации

Видно, что OllyDbg обнаружил цикл и, для удобства, свернул инструкции тела цикла в скобке.

Нажав правой кнопкой на EAX, можно выбрать «Follow in Dump» и позиция в окне памяти будет как раз там, где надо.

Здесь мы видим в памяти строку «hello!». После неё имеется как минимум 1 нулевой байт, затем случайный мусор. Если OllyDbg видит, что в регистре содержится адрес какой-то строки, он показывает эту строку.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Нажмем F8 (сделать шаг, не входя в функцию) столько раз, чтобы текущий адрес снова был в начале тела цикла:

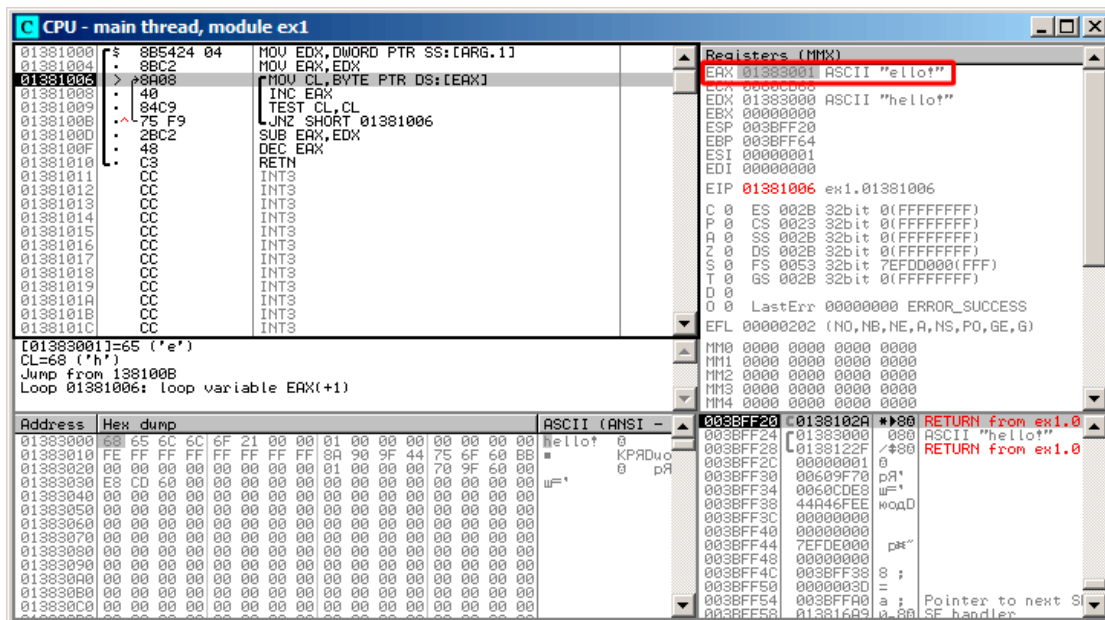


Рис. 1.59: OllyDbg: начало второй итерации

Видно, что EAX уже содержит адрес второго символа в строке.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Будем нажимать F8 достаточное количество раз, чтобы выйти из цикла:

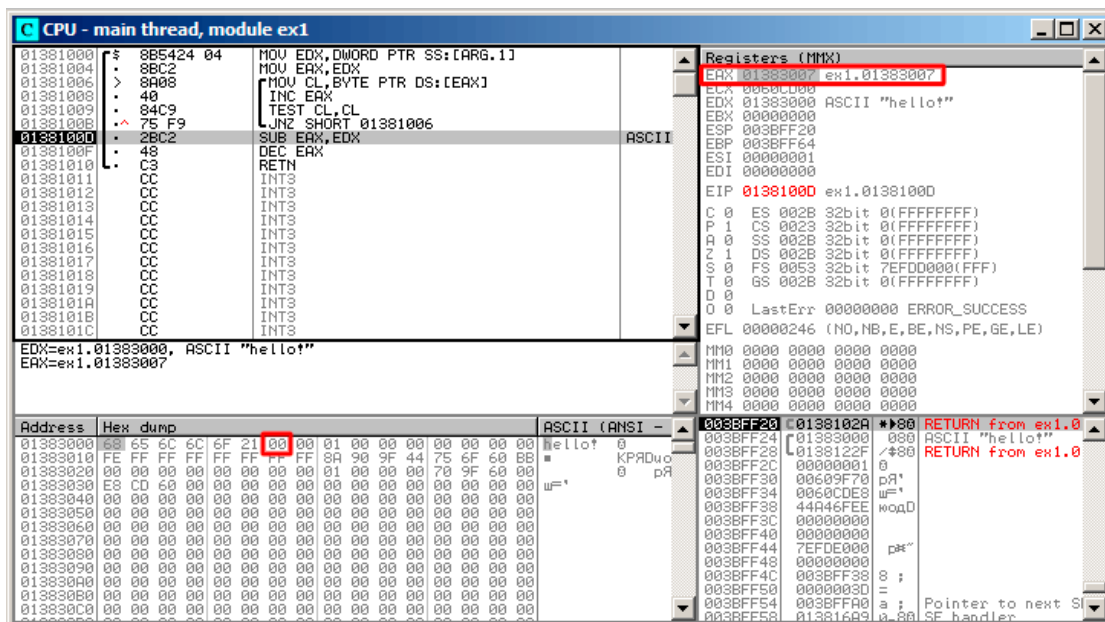


Рис. 1.60: OllyDbg: сейчас будет вычисление разницы указателей

Увидим, что EAX теперь содержит адрес нулевого байта, следующего сразу за строкой плюс 1 (потому что INC EAX исполнен вне зависимости от того, выходим мы из цикла, или нет).

A EDX так и не менялся — он всё ещё указывает на начало строки. Здесь сейчас будет вычисляться разница между этими двумя адресами.



Инструкция SUB исполнилась:

The screenshot shows the OllyDbg interface with the following details:

- Assembly View:**

```

01381000 8B5424 04 MOV EDX, DWORD PTR SS:[ARG.1]
01381004 8BC2 MOV EAX, EDX
01381006 8A08 MOV CL, BYTE PTR DS:[EAX]
01381008 40 INC EAX
01381009 84C9 TEST CL, CL
0138100B JNZ SHORT 01381006
0138100E 2BC2 SUB EAX, EDX
01381010 C3 RETN
01381011 CC INT3
01381012 CC INT3
01381013 CC INT3
01381014 CC INT3
01381015 CC INT3
01381016 CC INT3
01381017 CC INT3
01381018 CC INT3
01381019 CC INT3
0138101A CC INT3
0138101B CC INT3
0138101C CC INT3

```
- Registers (MMX):**

```

EAX: 00000007
ECX: 00000000
EDX: 01383000 ASCII "hello!"
EBX: 00000000
ESP: 003BFF20
EBP: 003BFF64
ESI: 00000001
EDI: 00000000
EIP: 0138100F ex1.0138100F
C 0 ES 002B 32bit 0(FFFFFFFF)
P 0 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 0 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 7EFD000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr: 00000000 ERROR_SUCCESS
EFL: 00000202 (NO, NB, NE, A, NS, PO, GE, G)
MM0: 0000 0000 0000 0000
MM1: 0000 0000 0000 0000
MM2: 0000 0000 0000 0000
MM3: 0000 0000 0000 0000
MM4: 0000 0000 0000 0000

```
- Hex dump:**

```

Address Hex dump ASCII (ANSI)
01383000 65 6C 6C 6F 21 00 00 01 00 00 00 00 00 00 00 hello!  
01383010 FE FF FF FF FF FF FF 3A 90 9F 44 75 6F 60 BE   KPADuo
01383020 00 00 00 00 00 00 00 01 00 00 00 00 70 9F 60 00     pA
01383030 E8 CD 60 00 00 00 00 00 00 00 00 00 00 00 00 00   ='
01383040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
01383050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
01383060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
01383070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
01383080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
01383090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
013830A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
013830B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
013830C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
003BFF20 0138102A *80 RETURN from ex1.0
003BFF24 01383000 088 ASCII "hello!"
003BFF28 0138122F /480 RETURN from ex1.0
003BFF2C 00000001  
003BFF30 00609F70 pA'
003BFF34 0060CDE8 w='
003BFF38 44A46FEE modD
003BFF3C 00000000  
003BFF40 00000000  
003BFF44 7EFD0000 pA"
003BFF48 00000000  
003BFF4C 003BFF38 ;
003BFF50 0000003D =
003BFF54 003BFFA0 a; Pointer to next SI
003BFF58 01381689 0-8A SF_handler

```

Рис. 1.61: OllyDbg: сейчас будет декремент EAX

Разница указателей сейчас в регистре EAX — 7.

Действительно, длина строки «hello!» — 6, но вместе с нулевым байтом — 7. Но `strlen()` должна возвращать количество ненулевых символов в строке. Так что сейчас будет исполняться декремент и выход из функции.

## Оптимизирующий GCC

Попробуем GCC 4.4.1 с включенной оптимизацией (ключ -O3):

```

public strlen
strlen      proc near
arg_0      = dword ptr 8

            push    ebp
            mov     ebp, esp
            mov     ecx, [ebp+arg_0]
            mov     eax, ecx

loc_8048418:
            movzx   edx, byte ptr [eax]
            add     eax, 1
            test    dl, dl

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](mailto:moi@adresa.ru). Спасибо!

```

                jnz     short loc_8048418
                not     ecx
                add     eax, ecx
                pop     ebp
                retn
strlen         endp

```

Здесь GCC не очень отстает от MSVC за исключением наличия MOVZX.

Впрочем, MOVZX здесь явно можно заменить на `mov dl, byte ptr [eax]`.

Но возможно, компилятору GCC просто проще помнить, что у него под переменную типа *char* отведен целый 32-битный регистр EDI и быть уверенным в том, что старшие биты регистра не будут замусорены.

Далее мы видим новую для нас инструкцию NOT. Эта инструкция инвертирует все биты в операнде. Можно сказать, что здесь это синонимично инструкции XOR ECX, 0fffffffh. NOT и следующая за ней инструкция ADD вычисляют разницу указателей и отнимают от результата единицу. Только происходит это слегка по-другому. Сначала ECX, где хранится указатель на *str*, инвертируется и от него отнимается единица.

Иными словами, в конце функции, после цикла, происходит примерно следующее:

```

ecx=str;
eax=eos;
ecx=(-ecx)-1;
eax=eax+ecx
return eax

```

... ЧТО ЭКВИВАЛЕНТНО:

```

ecx=str;
eax=eos;
eax=eax-ecx;
eax=eax-1;
return eax

```

Но почему GCC решил, что так будет лучше? Трудно угадать. Но наверное, оба эти варианта работают примерно одинаково в плане эффективности и скорости.

## ARM

### 32-битный ARM

#### Неоптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Листинг 1.190: Неоптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

```

_strlen
eos = -8
str = -4

    SUB    SP, SP, #8 ; выделить 8 байт для локальных переменных
    STR    R0, [SP,#8+str]
    LDR    R0, [SP,#8+str]
    STR    R0, [SP,#8+eos]

loc_2CB8 ; CODE XREF: _strlen+28
    LDR    R0, [SP,#8+eos]
    ADD    R1, R0, #1
    STR    R1, [SP,#8+eos]
    LDRSB  R0, [R0]
    CMP    R0, #0
    BEQ    loc_2CD4
    B      loc_2CB8
loc_2CD4 ; CODE XREF: _strlen+24
    LDR    R0, [SP,#8+eos]
    LDR    R1, [SP,#8+str]
    SUB    R0, R0, R1 ; R0=eos-str
    SUB    R0, R0, #1 ; R0=R0-1
    ADD    SP, SP, #8 ; освободить выделенные 8 байт
    BX    LR

```

Неоптимизирующий LLVM генерирует слишком много кода. Зато на этом примере можно посмотреть, как функции работают с локальными переменными в стеке.

В нашей функции только локальных переменных две — это два указателя: *eos* и *str*. В этом листинге сгенерированном при помощи **IDA!** мы переименовали *var\_8* и *var\_4* в *eos* и *str* вручную.

Итак, первые несколько инструкций просто сохраняют входное значение в обеих переменных *str* и *eos*.

С метки *loc\_2CB8* начинается тело цикла.

Первые три инструкции в теле цикла (LDR, ADD, STR) загружают значение *eos* в R0. Затем происходит инкремент значения и оно сохраняется в локальной переменной *eos* расположенной в стеке.

Следующая инструкция LDRSB R0, [R0] («Load Register Signed Byte») загружает байт из памяти по адресу R0, расширяет его до 32-бит считая его знаковым (signed) и сохраняет в R0 <sup>105</sup>. Это немного похоже на инструкцию MOVSH в x86. Компилятор считает этот байт знаковым (signed), потому что тип *char* по стандарту Си — знаковый.

Об этом уже было немного написано (?? (стр. ??)) в этой же секции, но посвященной x86.

<sup>105</sup>Компилятор Keil считает тип *char* знаковым, как и MSVC и GCC.

Следует также заметить, что в ARM нет возможности использовать 8-битную или 16-битную часть регистра, как это возможно в x86.

Вероятно, это связано с тем, что за x86 тянется длинный шлейф совместимости со своими предками, вплоть до 16-битного 8086 и даже 8-битного 8080, а ARM разрабатывался с чистого листа как 32-битный RISC-процессор.

Следовательно, чтобы работать с отдельными байтами на ARM, так или иначе придется использовать 32-битные регистры.

Итак, LDRSB загружает символы из строки в R0, по одному.

Следующие инструкции CMP и **BEQ!** проверяют, является ли этот символ 0.

Если не 0, то происходит переход на начало тела цикла. А если 0, выходим из цикла.

В конце функции вычисляется разница между *eos* и *str*, вычитается единица, и вычисленное значение возвращается через R0.

N.B. В этой функции не сохранялись регистры. По стандарту регистры R0-R3 называются также «scratch registers». Они предназначены для передачи аргументов и их значения не нужно восстанавливать при выходе из функции, потому что они больше не нужны в вызывающей функции. Таким образом, их можно использовать как захочется.

А так как никакие больше регистры не используются, то и сохранять нечего.

Поэтому управление можно вернуть вызывающей функции простым переходом (BX) по адресу в регистре **LR!**

### Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb)

Листинг 1.191: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb)

<code>_strlen</code>	<code>MOV</code>	<code>R1, R0</code>
<code>loc_2DF6</code>	<code>LDRB.W</code>	<code>R2, [R1], #1</code>
	<code>CMP</code>	<code>R2, #0</code>
	<code>BNE</code>	<code>loc_2DF6</code>
	<code>MVNS</code>	<code>R0, R0</code>
	<code>ADD</code>	<code>R0, R1</code>
	<code>BX</code>	<code>LR</code>

Оптимизирующий LLVM решил, что под переменные *eos* и *str* выделять место в стеке не обязательно, и эти переменные можно хранить прямо в регистрах.

Перед началом тела цикла *str* будет находиться в R0, а *eos* — в R1.

Инструкция `LDRB.W R2, [R1], #1` загружает в R2 байт из памяти по адресу R1, расширяя его как знаковый (signed), до 32-битного значения, но не только это.

#1 в конце инструкции означает «Post-indexed addressing», т.е. после загрузки байта к R1 добавится единица.

Читайте больше об этом: ?? (стр. ??).

Далее в теле цикла можно увидеть CMP и **BNE!**<sup>106</sup>. Они продолжают работу цикла до тех пор, пока не будет встречен 0.

После конца цикла MVNS<sup>107</sup> (инвертирование всех бит, NOT в x86) и ADD вычисляют  $eos - str - 1$ . На самом деле, эти две инструкции вычисляют  $R0 = str + eos$ , что эквивалентно тому, что было в исходном коде. Почему это так, уже было описано чуть раньше, здесь (?? (стр. ??)).

Вероятно, LLVM, как и GCC, посчитал, что такой код может быть короче (или быстрее).

### Оптимизирующий Keil 6/2013 (Режим ARM)

Листинг 1.192: Оптимизирующий Keil 6/2013 (Режим ARM)

```

_strlen
    MOV    R1, R0

loc_2C8
    LDRB  R2, [R1],#1
    CMP   R2, #0
    SUBEQ R0, R1, R0
    SUBEQ R0, R0, #1
    BNE   loc_2C8
    BX    LR

```

Практически то же самое, что мы уже видели, за тем исключением, что выражение  $str - eos - 1$  может быть вычислено не в самом конце функции, а прямо в теле цикла.

Суффикс -EQ означает, что инструкция будет выполнена только если операнды в исполненной перед этим инструкции CMP были равны.

Таким образом, если в R0 будет 0, обе инструкции SUBEQ исполнятся и результат останется в R0.

### ARM64

### Оптимизирующий GCC (Linaro) 4.9

```

my_strlen:
    mov    x1, x0

```

<sup>106</sup>**BNE!**

<sup>107</sup>MoVe Not

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; X1 теперь временный регистр (eos), работающий, как курсор
.L58:
; загрузить байт из X1 в W2, инкремент X1 (пост-индекс)
ldrb    w2, [x1],1
; Compare and Branch if NonZero: сравнить W0 с нулем,
; перейти на .L58 если не ноль
cbnz    w2, .L58
; вычислить разницу между изначальным указателем
; в X0 и текущим адресом в X1
sub     x0, x1, x0
; декремент младших 32-х бит результата
sub     w0, w0, #1
ret

```

Алгоритм такой же как и в ?? (стр. ??): найти нулевой байт, затем вычислить разницу между указателями, затем отнять 1 от результата. Комментарии добавлены автором книги.

Стоит добавить, что наш пример имеет ошибку: `my_strlen()` возвращает 32-битный `int`, тогда как должна возвращать `size_t` или иной 64-битный тип.

Причина в том, что теоретически, `strlen()` можно вызывать для огромных блоков в памяти, превышающих 4GB, так что она должна иметь возможность вернуть 64-битное значение на 64-битной платформе.

Так что из-за моей ошибки, последняя инструкция SUB работает над 32-битной частью регистра, тогда как предпоследняя SUB работает с полными 64-битными частями (она вычисляет разницу между указателями).

Это моя ошибка, но лучше оставить это как есть, как пример кода, который возможен в таком случае.

## Неоптимизирующий GCC (Linaro) 4.9

```

my_strlen:
; пролог функции
sub     sp, sp, #32
; первый аргумент (str) будет записан в [sp,8]
str     x0, [sp,8]
ldr     x0, [sp,8]
; скопировать переменную "str" в "eos"
str     x0, [sp,24]
nop
.L62:
; eos++
ldr     x0, [sp,24] ; загрузить "eos" в X0
add     x1, x0, 1   ; инкремент X0
str     x1, [sp,24] ; сохранить X0 в "eos"
; загрузить байт из памяти по адресу в X0 в W0
ldrb    w0, [x0]
; это ноль? (WZR это 32-битный регистр всегда содержащий ноль)
cmp     w0, wzr

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; переход если не ноль (Branch Not Equal)
    bne    .L62
; найден нулевой байт. вычисляем разницу.
; загрузить "eos" в X1
    ldr    x1, [sp,24]
; загрузить "str" в X0
    ldr    x0, [sp,8]
; вычислить разницу
    sub    x0, x1, x0
; декремент результата
    sub    w0, w0, #1
; эпилог функции
    add    sp, sp, 32
    ret

```

Более многословно. Переменные часто сохраняются в память и загружаются назад (локальный стек). Здесь та же ошибка: операция декремента происходит над 32-битной частью регистра.

## MIPS

Листинг 1.193: Оптимизирующий GCC 4.4.5 (IDA)

```

my_strlen:
; переменная "eos" всегда будет находиться в $v1:
    move   $v1, $a0

loc_4:
; загрузить байт по адресу в "eos" в $a1:
    lb     $a1, 0($v1)
    or     $at, $zero ; load delay slot, NOP
; если загруженный байт не ноль, перейти на loc_4:
    bnez   $a1, loc_4
; в любом случае, инкрементируем "eos":
    addiu  $v1, 1 ; branch delay slot
; цикл закончен. инвертируем переменную "str":
    nor    $v0, $zero, $a0
; $v0=-str-1
    jr     $ra
; возвращаемое значение = $v1 + $v0 = eos + ( -str-1 ) = eos - str - 1
    addu   $v0, $v1, $v0 ; branch delay slot

```

В MIPS нет инструкции NOT, но есть NOR — операция OR + NOT.

Эта операция широко применяется в цифровой электронике<sup>108</sup>. Например, космический компьютер Apollo Guidance Computer использовавшийся в программе «Аполлон» был построен исключительно на 5600 элементах NOR: [Jens Eickhoff, *Onboard Computers, Onboard Software and Satellite Operations: An Introduction*, (2011)]. Но элемент NOR не очень популярен в программировании.

Так что операция NOT реализована здесь как NOR DST, \$ZERO, SRC.

<sup>108</sup>NOR называют «универсальным элементом»

(добитовое инвертирование знакового числа это то же что и смена его знака с вычитанием 1 из результата.)

Так что NOT берет значение *str* и трансформирует его в  $-str - 1$ .

Следующая операция сложения готовит результат.

## 1.24. Замена одних арифметических инструкций на другие

В целях оптимизации одна инструкция может быть заменена другой, или даже группой инструкций. Например, ADD и SUB могут заменять друг друга: строка 18 в листинг.??.

Более того, не всегда замена тривиальна. Инструкция LEA, несмотря на оригинальное назначение, нередко применяется для простых арифметических действий: ?? (стр. ??).

### 1.24.1. Умножение

#### Умножение при помощи сложения

Вот простой пример:

```
unsigned int f(unsigned int a)
{
    return a*8;
};
```

Умножение на 8 заменяется на три инструкции сложения, делающих то же самое. Должно быть, оптимизатор в MSVC решил, что этот код может быть быстрее.

Листинг 1.194: Оптимизирующий MSVC 2010

```
_TEXT SEGMENT
_a$ = 8 ; size = 4
_f PROC
    mov     eax, DWORD PTR _a$[esp-4]
    add     eax, eax
    add     eax, eax
    add     eax, eax
    ret     0
_f ENDP
_TEXT ENDS
END
```

#### Умножение при помощи сдвигов

Ещё очень часто умножения и деления на числа вида  $2^n$  заменяются на инструкции сдвигов.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

unsigned int f(unsigned int a)
{
    return a*4;
};

```

Листинг 1.195: Неоптимизирующий MSVC 2010

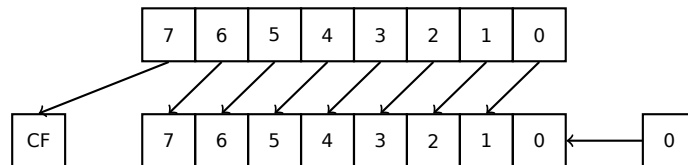
```

_a$ = 8          ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    shl    eax, 2
    pop     ebp
    ret     0
_f ENDP

```

Умножить на 4 это просто сдвинуть число на 2 бита влево, вставив 2 нулевых бита справа (как два самых младших бита). Это как умножить 3 на 100 — нужно просто дописать два нуля справа.

Вот как работает инструкция сдвига влево:



Добавленные биты справа — всегда нули.

Умножение на 4 в ARM:

Листинг 1.196: Неоптимизирующий Keil 6/2013 (Режим ARM)

```

f PROC
    LSL    r0, r0, #2
    BX    lr
    ENDP

```

Умножение на 4 в MIPS:

Листинг 1.197: Оптимизирующий GCC 4.4.5 (IDA)

```

jr    $ra
sll   $v0, $a0, 2 ; branch delay slot

```

SLL это «Shift Left Logical».

### Умножение при помощи сдвигов, сложений и вычитаний

Можно избавиться от операции умножения, если вы умножаете на числа вроде 7 или 17, и использовать сдвиги.

Здесь используется относительно простая математика.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

**32-бита**

```
#include <stdint.h>

int f1(int a)
{
    return a*7;
};

int f2(int a)
{
    return a*28;
};

int f3(int a)
{
    return a*17;
};
```

**x86**

Листинг 1.198: Оптимизирующий MSVC 2012

```
; a*7
_a$ = 8
_f1 PROC
    mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
    lea    eax, DWORD PTR [ecx*8]
; EAX=ECX*8
    sub    eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
    ret    0
_f1     ENDP

; a*28
_a$ = 8
_f2 PROC
    mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
    lea    eax, DWORD PTR [ecx*8]
; EAX=ECX*8
    sub    eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
    shl   eax, 2
; EAX=EAX<<2=(a*7)*4=a*28
    ret    0
_f2     ENDP

; a*17
_a$ = 8
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

_f3 PROC
mov     eax, DWORD PTR _a$[esp-4]
; EAX=a
shl     eax, 4
; EAX=EAX<<4=EAX*16=a*16
add     eax, DWORD PTR _a$[esp-4]
; EAX=EAX+a=a*16+a=a*17
ret     0
_f3 ENDP

```

## ARM

Keil, генерируя код для режима ARM, использует модификаторы инструкции, в которых можно задавать сдвиг для второго операнда:

Листинг 1.199: Оптимизирующий Keil 6/2013 (Режим ARM)

```

; a*7
||f1|| PROC
RSB     r0, r0, r0, LSL #3
; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
BX      lr
ENDP

; a*28
||f2|| PROC
RSB     r0, r0, r0, LSL #3
; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
LSL     r0, r0, #2
; R0=R0<<2=R0*4=a*7*4=a*28
BX      lr
ENDP

; a*17
||f3|| PROC
ADD     r0, r0, r0, LSL #4
; R0=R0+R0<<4=R0+R0*16=R0*17=a*17
BX      lr
ENDP

```

Но таких модификаторов в режиме Thumb нет.

И он также не смог оптимизировать функцию f2():

Листинг 1.200: Оптимизирующий Keil 6/2013 (Режим Thumb)

```

; a*7
||f1|| PROC
LSLS    r1, r0, #3
; R1=R0<<3=a<<3=a*8
SUBS    r0, r1, r0
; R0=R1-R0=a*8-a=a*7
BX      lr

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        ENDP

; a*28
||f2|| PROC
        MOVS    r1,#0x1c ; 28
; R1=28
        MULS    r0,r1,r0
; R0=R1*R0=28*a
        BX      lr
        ENDP

; a*17
||f3|| PROC
        LSLS    r1,r0,#4
; R1=R0<<4=R0*16=a*16
        ADDS    r0,r0,r1
; R0=R0+R1=a+a*16=a*17
        BX      lr
        ENDP

```

## MIPS

Листинг 1.201: Оптимизирующий GCC 4.4.5 (IDA)

```

_f1:
        sll     $v0, $a0, 3
; $v0 = $a0<<3 = $a0*8
        jr      $ra
        subu    $v0, $a0 ; branch delay slot
; $v0 = $v0-$a0 = $a0*8-$a0 = $a0*7

_f2:
        sll     $v0, $a0, 5
; $v0 = $a0<<5 = $a0*32
        sll     $a0, 2
; $a0 = $a0<<2 = $a0*4
        jr      $ra
        subu    $v0, $a0 ; branch delay slot
; $v0 = $a0*32-$a0*4 = $a0*28

_f3:
        sll     $v0, $a0, 4
; $v0 = $a0<<4 = $a0*16
        jr      $ra
        addu    $v0, $a0 ; branch delay slot
; $v0 = $a0*16+$a0 = $a0*17

```

## 64-бита

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
#include <stdint.h>

int64_t f1(int64_t a)
{
    return a*7;
};

int64_t f2(int64_t a)
{
    return a*28;
};

int64_t f3(int64_t a)
{
    return a*17;
};
```

**x64**

Листинг 1.202: Оптимизирующий MSVC 2012

```
; a*7
f1:
    lea    rax, [0+rdi*8]
; RAX=RDI*8=a*8
    sub    rax, rdi
; RAX=RAX-RDI=a*8-a=a*7
    ret

; a*28
f2:
    lea    rax, [0+rdi*4]
; RAX=RDI*4=a*4
    sal    rdi, 5
; RDI=RDI<<5=RDI*32=a*32
    sub    rdi, rax
; RDI=RDI-RAX=a*32-a*4=a*28
    mov    rax, rdi
    ret

; a*17
f3:
    mov    rax, rdi
    sal    rax, 4
; RAX=RAX<<4=a*16
    add    rax, rdi
; RAX=a*16+a=a*17
    ret
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

**ARM64**

GCC 4.9 для ARM64 также очень лаконичен благодаря модификаторам сдвига:

Листинг 1.203: Оптимизирующий GCC (Linaro) 4.9 ARM64

```

; a*7
f1:
    lsl    x1, x0, 3
; X1=X0<<3=X0*8=a*8
    sub   x0, x1, x0
; X0=X1-X0=a*8-a=a*7
    ret

; a*28
f2:
    lsl    x1, x0, 5
; X1=X0<<5=a*32
    sub   x0, x1, x0, lsl 2
; X0=X1-X0<<2=a*32-a<<2=a*32-a*4=a*28
    ret

; a*17
f3:
    add   x0, x0, x0, lsl 4
; X0=X0+X0<<4=a+a*16=a*17
    ret

```

**Алгоритм умножения Бута**

Когда-то компьютеры были большими и дорогими настолько, что некоторые не имели поддержки операции умножения в **CPU!**, например Data General Nova. И когда операция умножения была нужна, она обеспечивалась программно, например, при помощи алгоритма Бута (*Booth's multiplication algorithm*). Это алгоритм перемножения чисел используя только операции сложения и сдвига.

То что ныне делают компиляторы для оптимизации — это не совсем то, но цель (умножение) и средства (замена более быстрыми операциями) те же.

**1.24.2. Деление****Деление используя сдвиги**

Например, возьмем деление на 4:

```

unsigned int f(unsigned int a)
{
    return a/4;
};

```

Имеем в итоге (MSVC 2010):

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Листинг 1.204: MSVC 2010

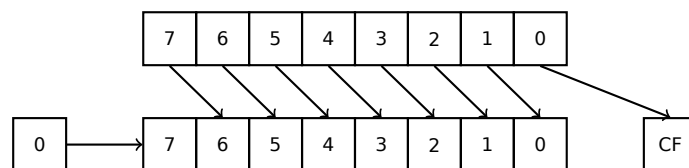
```

_a$ = 8      ; size = 4
_f PROC
    mov     eax, DWORD PTR _a$[esp-4]
    shr     eax, 2
    ret     0
_f ENDP

```

Инструкция SHR (*SHift Right*) в данном примере сдвигает число на 2 бита вправо. При этом освободившиеся два бита слева (т.е. самые старшие разряды) выставляются в нули. А самые младшие 2 бита выкидываются. Фактически, эти два выкинутых бита — остаток от деления.

Инструкция SHR работает так же как и SHL, только в другую сторону.



Для того, чтобы это проще понять, представьте себе десятичную систему счисления и число 23. 23 можно разделить на 10 просто откинув последний разряд (3 — это остаток от деления). После этой операции останется 2 как **частное**.

Так что остаток выбрасывается, но это нормально, мы все-таки работаем с целочисленными значениями, а не с **вещественными!**

Деление на 4 в ARM:

Листинг 1.205: Неоптимизирующий Keil 6/2013 (Режим ARM)

```

f PROC
    LSR     r0, r0, #2
    BX     lr
    ENDP

```

Деление на 4 в MIPS:

Листинг 1.206: Оптимизирующий GCC 4.4.5 (IDA)

```

jr     $ra
srl   $v0, $a0, 2 ; branch delay slot

```

Инструкция SRL это «Shift Right Logical».

### 1.24.3. Упражнение

- <http://challenges.re/59>

## 1.25. Работа с FPU

**FPU!**<sup>109</sup> — блок в процессоре работающий с числами с плавающей запятой.

Раньше он назывался «сопроцессором» и он стоит немного в стороне от **CPU!**.

### 1.25.1. IEEE 754

Число с плавающей точкой в формате IEEE 754 состоит из *знака, мантииссы*<sup>110</sup> и *экспоненты*.

### 1.25.2. x86

Перед изучением **FPU!** в x86 полезно ознакомиться с тем как работают стековые машины или ознакомиться с основами языка Forth.

Интересен факт, что в свое время (до 80486) сопроцессор был отдельным чипом на материнской плате, и вследствие его высокой цены, он не всегда присутствовал. Его можно было докупить и установить отдельно<sup>111</sup>. Начиная с 80486 DX в состав процессора всегда входит FPU.

Этот факт может напоминать такой рудимент как наличие инструкции FWAIT, которая заставляет **CPU!** ожидать, пока **FPU!** закончит работу. Другой рудимент это тот факт, что опкоды **FPU!**-инструкций начинаются с т.н. «escape»-опкодов (D8..DF) как опкоды, передающиеся в отдельный сопроцессор.

FPU имеет стек из восьми 80-битных регистров: ST(0)..ST(7). Для краткости, **IDA!** и OllyDbg отображают ST(0) как ST, что в некоторых учебниках и документах означает «Stack Top» («вершина стека»). Каждый регистр может содержать число в формате IEEE 754.

### 1.25.3. ARM, MIPS, x86/x64 SIMD

В ARM и MIPS FPU это не стек, а просто набор регистров, к которым можно обращаться произвольно, как к **GPR!**.

Такая же идеология применяется в расширениях SIMD в процессорах x86/x64.

### 1.25.4. Си/Си++

В стандартных Си/Си++ имеются два типа для работы с числами с плавающей запятой: *float* (число одинарной точности, 32 бита)<sup>112</sup> и *double* (число двойной точности, 64 бита).

<sup>109</sup>**FPU!**

<sup>110</sup>*significand* или *fraction* в англоязычной литературе

<sup>111</sup>Например, Джон Кармак использовал в своей игре Doom числа с фиксированной запятой, хранящиеся в обычных 32-битных **GPR!** (16 бит на целую часть и 16 на дробную), чтобы Doom работал на 32-битных компьютерах без FPU, т.е. 80386 и 80486 SX.

<sup>112</sup>Формат представления чисел с плавающей точкой одинарной точности затрагивается в разделе *Работа с типом float как со структурой (?? (стр. ??))*.



В [Donald E. Knuth, *The Art of Computer Programming, Volume 2, 3rd ed.*, (1997)246] мы можем найти что *single-precision* означает, что значение с плавающей точкой может быть помещено в одно [32-битное] машинное слово, а *double-precision* означает, что оно размещено в двух словах (64 бита).

GCC также поддерживает тип *long double (extended precision, 80 бит)*, но MSVC — нет.

Несмотря на то, что *float* занимает столько же места, сколько и *int* на 32-битной архитектуре, представление чисел, разумеется, совершенно другое.

### 1.25.5. Простой пример

Рассмотрим простой пример:

```
#include <stdio.h>

double f (double a, double b)
{
    return a/3.14 + b*4.1;
};

int main()
{
    printf ("%f\n", f(1.2, 3.4));
};
```

#### x86

#### MSVC

Компилируем в MSVC 2010:

Листинг 1.207: MSVC 2010: f ()

```
CONST    SEGMENT
__real@4010666666666666 DQ 0401066666666666r    ; 4.1
CONST    ENDS
CONST    SEGMENT
__real@40091eb851eb851f DQ 040091eb851eb851fr    ; 3.14
CONST    ENDS
_TEXT    SEGMENT
_a$ = 8          ; size = 8
_b$ = 16         ; size = 8
_f PROC
    push    ebp
    mov     ebp, esp
    fld     QWORD PTR _a$[ebp]

; текущее состояние стека: ST(0) = _a

    fdiv   QWORD PTR __real@40091eb851eb851f
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; текущее состояние стека: ST(0) = результат деления _a на 3.14

fld    QWORD PTR _b$[ebp]

; текущее состояние стека: ST(0) = _b;
; ST(1) = результат деления _a на 3.14

fmul   QWORD PTR __real@4010666666666666

; текущее состояние стека:
; ST(0) = результат умножения _b на 4.1;
; ST(1) = результат деления _a на 3.14

faddp  ST(1), ST(0)

; текущее состояние стека: ST(0) = результат сложения

pop    ebp
ret    0
_f    ENDP

```

FLD берет 8 байт из стека и загружает их в регистр  $ST(0)$ , автоматически конвертируя во внутренний 80-битный формат (*extended precision*).

FDIV делит содержимое регистра  $ST(0)$  на число, лежащее по адресу `__real@40091eb851eb851f` — там закодировано значение 3,14. Синтаксис ассемблера не поддерживает подобные числа, поэтому мы там видим шестнадцатеричное представление числа 3,14 в формате IEEE 754.

После выполнения FDIV в  $ST(0)$  остается **частное**.

Кстати, есть ещё инструкция FDIVP, которая делит  $ST(1)$  на  $ST(0)$ , выталкивает эти числа из стека и заталкивает результат. Если вы знаете язык Forth, то это как раз оно и есть — стековая машина.

Следующая FLD заталкивает в стек значение  $b$ .

После этого в  $ST(1)$  перемещается результат деления, а в  $ST(0)$  теперь  $b$ .

Следующий FMUL умножает  $b$  из  $ST(0)$  на значение `__real@4010666666666666` — там лежит число 4,1 — и оставляет результат в  $ST(0)$ .

Самая последняя инструкция FADDP складывает два значения из вершины стека в  $ST(1)$  и затем выталкивает значение, лежащее в  $ST(0)$ . Таким образом результат сложения остается на вершине стека в  $ST(0)$ .

Функция должна вернуть результат в  $ST(0)$ , так что больше ничего здесь не производится, кроме эпилога функции.

## MSVC + OllyDbg

2 пары 32-битных слов обведены в стеке красным. Каждая пара — это числа двойной точности в формате IEEE 754, переданные из main().

Видно, как первая FLD загружает значение 1,2 из стека и помещает в регистр ST(0):

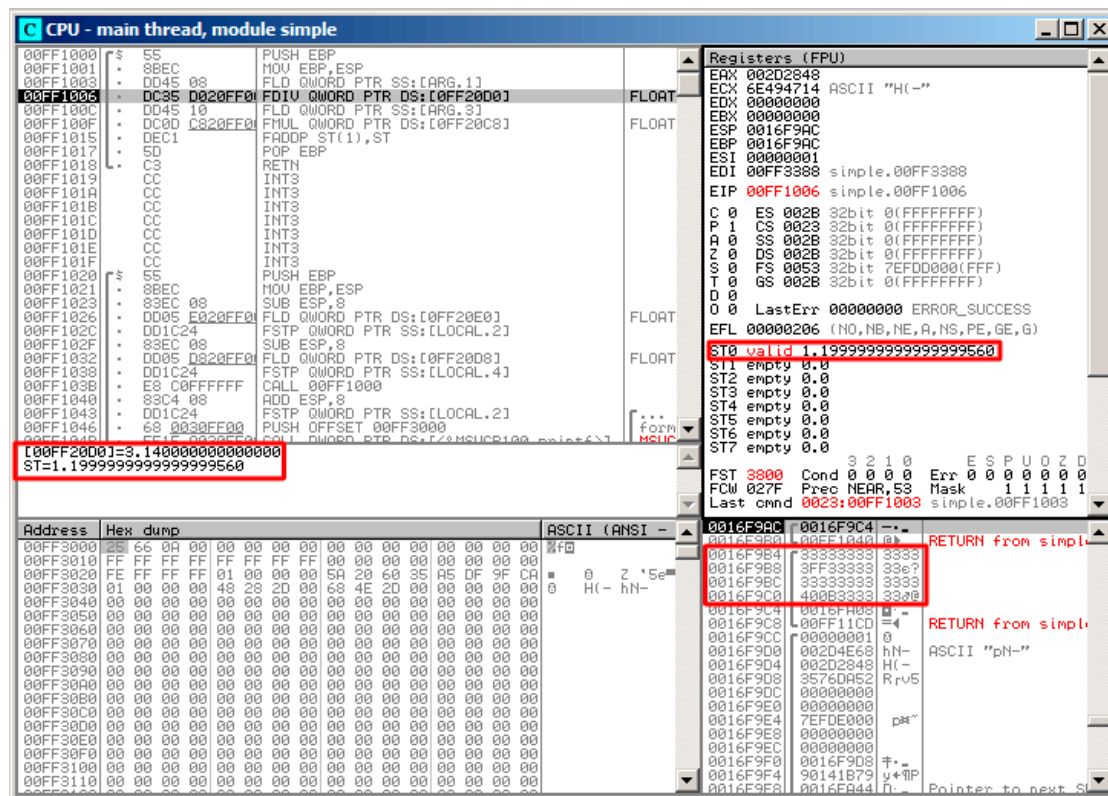


Рис. 1.62: OllyDbg: первая FLD исполнялась

Из-за неизбежных ошибок конвертирования числа из 64-битного IEEE 754 в 80-битное (внутреннее в FPU), мы видим здесь 1,1999..., что очень близко к 1,2.

Прямо сейчас EIP указывает на следующую инструкцию (FDIV), загружающую константу двойной точности из памяти.

Для удобства, OllyDbg показывает её значение: 3,14.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Трассируем дальше. FDIV исполнилась, теперь ST(0) содержит 0,382...(частное):

The screenshot shows the OllyDbg interface with the following details:

- CPU - main thread, module simple:**
  - Address 00FF1000: 55 PUSH EBP
  - Address 00FF1001: 8BEC MOV EBP,ESP
  - Address 00FF1003: DD45 08 FLD QWORD PTR SS:[ARG.1]
  - Address 00FF1006: DC35 0020FF00 FLD QWORD PTR DS:[0FF20D0]
  - Address 00FF1009: DD45 10 FLD QWORD PTR SS:[ARG.3]
  - Address 00FF100F: DC00 C020FF00 FLD QWORD PTR DS:[0FF20C8]
  - Address 00FF1015: DEC1 FADD ST(1),ST
  - Address 00FF1017: 5D POP EBP
  - Address 00FF1018: C3 RETN
  - Address 00FF1019: CC INT3
  - Address 00FF101A: CC INT3
  - Address 00FF101B: CC INT3
  - Address 00FF101C: CC INT3
  - Address 00FF101E: CC INT3
  - Address 00FF101F: CC INT3
  - Address 00FF1020: 55 PUSH EBP
  - Address 00FF1021: 8BEC MOV EBP,ESP
  - Address 00FF1023: 83EC 08 SUB ESP,8
  - Address 00FF1026: DD05 F020FF00 FLD QWORD PTR DS:[0FF20E0]
  - Address 00FF102C: DD1C24 FSTP QWORD PTR SS:[LOCAL.2]
  - Address 00FF102F: 83EC 08 SUB ESP,8
  - Address 00FF1032: DD05 0820FF00 FLD QWORD PTR DS:[0FF20D8]
  - Address 00FF1038: DD1C24 FSTP QWORD PTR SS:[LOCAL.4]
  - Address 00FF103B: E8 C0FFFFFF CALL 00FF1000
  - Address 00FF1040: 83C4 08 ADD ESP,8
  - Address 00FF1043: DD1C24 FSTP QWORD PTR SS:[LOCAL.2]
  - Address 00FF1046: 68 0030FE00 PUSH OFFSET 00FF3000
  - Address 00FF1048: E4 0030FE00 CALL QWORD PTR DS:[MSUCR100.point63]
- Registers (FPU):**
  - EAX: 002D2848
  - ECX: 6E494714 ASCII "H(-"
  - EDX: 00000000
  - EBX: 00000000
  - ESP: 0016F9AC
  - EBP: 0016F9AC
  - ESI: 00000001
  - EDI: 00FF3338 simple.00FF3338
  - EIP: 00FF100C simple.00FF100C
  - C 0: ES 002B 32bit 0(FFFFFFFF)
  - P 1: CS 0023 32bit 0(FFFFFFFF)
  - A 0: SS 002B 32bit 0(FFFFFFFF)
  - Z 0: DS 002B 32bit 0(FFFFFFFF)
  - S 0: FS 0053 32bit 7EFD0000(FFF)
  - T 0: GS 002B 32bit 0(FFFFFFFF)
  - D 0
  - O 0 LastErr: 00000000 ERROR\_SUCCESS
  - EFL: 00000206 (NO,NB,NE,A,NS,PE,GE,G)
  - ST0: valid 0.382165605095413719
  - ST1: empty 0.0
  - ST2: empty 0.0
  - ST3: empty 0.0
  - ST4: empty 0.0
  - ST5: empty 0.0
  - ST6: empty 0.0
  - ST7: empty 0.0
- Stack [0016F9C1]=3.4000000000000000:**
  - Address 00FF3000: 25 66 0A 00 00 00 00 00 00 00 00 00 00 00 00 00
  - Address 00FF3010: FF FF FF FF FF FF FF FF 00 00 00 00 00 00 00 00
  - Address 00FF3020: FE FF FF FF 01 00 00 00 5A 20 60 35 A5 DF 9F CA
  - Address 00FF3030: 01 00 00 00 43 28 2D 00 68 4E 2D 00 00 00 00 00
  - Address 00FF3040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - Address 00FF3050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - Address 00FF3060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - Address 00FF3070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - Address 00FF3080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - Address 00FF3090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - Address 00FF30A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - Address 00FF30B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - Address 00FF30C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - Address 00FF30D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - Address 00FF30E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - Address 00FF30F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - Address 00FF3100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - Address 00FF3110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Рис. 1.63: OllyDbg: FDIV исполнилась

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



Затем: FMUL исполнилась, теперь в ST(0) произведение:

The screenshot shows the OllyDbg interface with the following details:

- CPU - main thread, module simple:**
  - Address 00FF1015: `DEC1` instruction is highlighted.
  - Registers (FPU) window shows:
    - ST0 valid **13.939999999999997730** (highlighted in red)
    - ST1 valid **0.3821656050955413719**
- Memory Dump:**
  - Address 00FF3000: `25 66 0A 00 00 00 00 00 00 00 00 00 00 00 00 00`
  - Address 00FF3010: `FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF`
  - Address 00FF3020: `FE FF FF FF 01 00 00 00 5A 20 60 35 A5 DF 9F CA`
  - Address 00FF3030: `01 00 00 00 48 28 2D 00 68 4E 2D 00 00 00 00 00`
  - Address 00FF3040: `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`
  - Address 00FF3050: `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`
  - Address 00FF3060: `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`
  - Address 00FF3070: `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`
  - Address 00FF3080: `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`
  - Address 00FF3090: `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`
  - Address 00FF30A0: `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`
  - Address 00FF30B0: `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`
  - Address 00FF30C0: `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`
  - Address 00FF30D0: `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`
  - Address 00FF30E0: `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`
  - Address 00FF30F0: `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`
  - Address 00FF3100: `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`
  - Address 00FF3110: `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`

Рис. 1.65: OllyDbg: FMUL исполнилась

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Затем: FADDP исполнилась, теперь в ST(0) сумма, а ST(1) очистился:

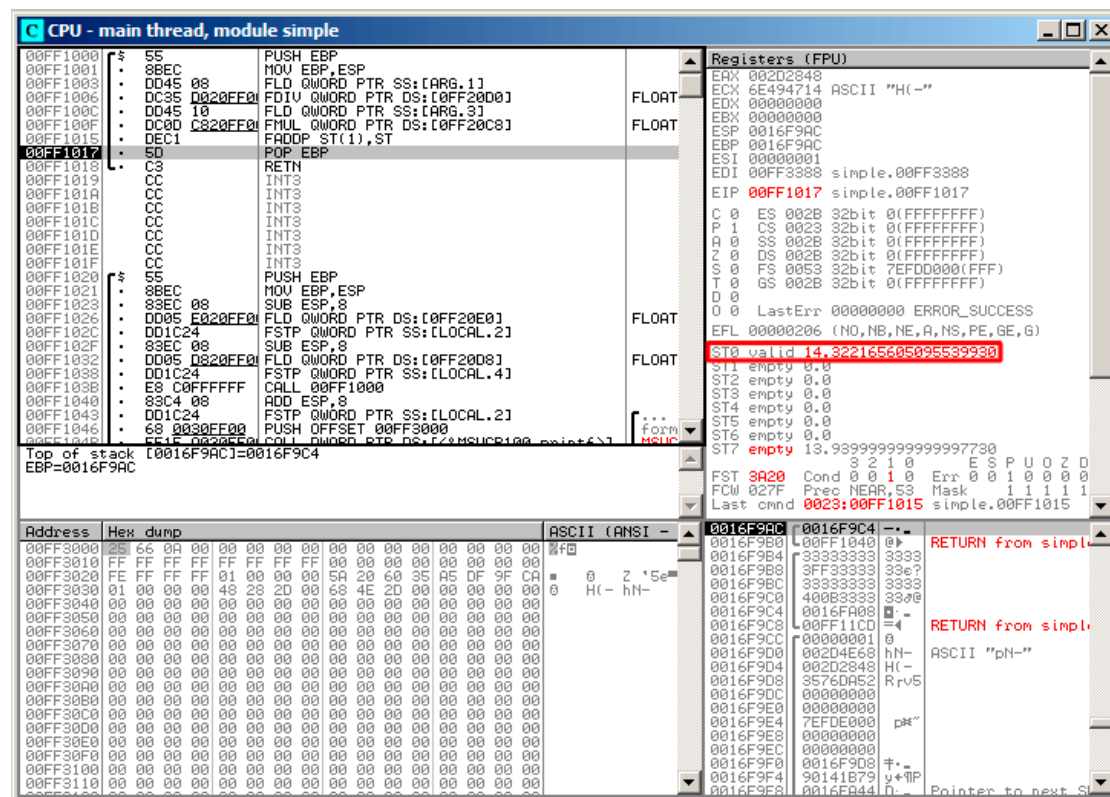


Рис. 1.66: OllyDbg: FADDP исполнилась

Сумма остается в ST(0) потому что функция возвращает результат своей работы через ST(0).

Позже main() возьмет это значение оттуда.

Мы также видим кое-что необычное: значение 13,93...теперь находится в ST(7).

Почему?

Мы читали в этой книге, что регистры в **FPU!** представляют собой стек: ?? (стр. ??). Но это упрощение. Представьте, если бы в железе было бы так, как описано. Тогда при каждом заталкивании (или выталкивании) в стек, все остальные 7 значений нужно было бы передвигать (или копировать) в соседние регистры, а это слишком затратно.

Так что в реальности у **FPU!** есть просто 8 регистров и указатель (называемый TOP), содержащий номер регистра, который в текущий момент является «вершиной стека».

При заталкивании значения в стек регистр TOP меняется, и указывает на свободный регистр. Затем значение записывается в этот регистр.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



При выталкивании значения из стека процедура обратная. Однако освобожденный регистр не обнуляется (наверное, можно было бы сделать, чтобы обнулялся, но это лишняя работа и работало бы медленнее). Так что это мы здесь и видим. Можно сказать, что FADDP сохранила сумму, а затем вытолкнула один элемент.

Но в реальности, эта инструкция сохранила сумму и затем передвинула регистр TOP.

Было бы ещё точнее сказать, что регистры **FPU!** представляют собой кольцевой буфер.

## GCC

GCC 4.4.1 (с опцией -O3) генерирует похожий код, хотя и с некоторой разницей:

Листинг 1.208: Оптимизирующий GCC 4.4.1

```

public f
proc near
f
    arg_0      = qword ptr 8
    arg_8      = qword ptr 10h

    push     ebp
    fld     ds:dbl_8048608 ; 3.14

; состояние стека сейчас: ST(0) = 3.14

    mov     ebp, esp
    fdivr   [ebp+arg_0]

; состояние стека сейчас: ST(0) = результат деления

    fld     ds:dbl_8048610 ; 4.1

; состояние стека сейчас: ST(0) = 4.1, ST(1) = результат деления

    fmul    [ebp+arg_8]

; состояние стека сейчас: ST(0) = результат умножения, ST(1) = результат
; деления

    pop     ebp
    faddp   st(1), st

; состояние стека сейчас: ST(0) = результат сложения

    retn
f
endp

```

Разница в том, что в стек сначала заталкивается 3,14 (в ST(0)), а затем значение из arg\_0 делится на то, что лежит в регистре ST(0).

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



FDIVR означает *Reverse Divide* — делить, поменяв делитель и делимое места. Точно такой же инструкции для умножения нет, потому что она была бы бессмысленна (ведь умножение операция коммутативная), так что остается только FMUL без соответствующей ей -R инструкции.

FADDP не только складывает два значения, но также и выталкивает из стека одно значение. После этого в ST(0) остается только результат сложения.

### ARM: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

Пока в ARM не было стандартного набора инструкций для работы с числами с плавающей точкой, разные производители процессоров могли добавлять свои расширения для работы с ними. Позже был принят стандарт VFP (*Vector Floating Point*).

Важное отличие от x86 в том, что там вы работаете с FPU-стеком, а здесь стека нет, вы работаете просто с регистрами.

Листинг 1.209: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

```
f
    VLDR        D16, =3.14
    VMOV        D17, R0, R1 ; загрузить "a"
    VMOV        D18, R2, R3 ; загрузить "b"
    VDIV.F64    D16, D17, D16 ; a/3.14
    VLDR        D17, =4.1
    VMUL.F64    D17, D18, D17 ; b*4.1
    VADD.F64    D16, D17, D16 ; +
    VMOV        R0, R1, D16
    BX          LR

dbl_2C98      DCFD 3.14          ; DATA XREF: f
dbl_2CA0      DCFD 4.1          ; DATA XREF: f+10
```

Итак, здесь мы видим использование новых регистров с префиксом D.

Это 64-битные регистры. Их 32 и их можно использовать для чисел с плавающей точкой двойной точности (double) и для SIMD (в ARM это называется NEON).

Имеются также 32 32-битных S-регистра. Они применяются для работы с числами с плавающей точкой одинарной точности (float).

Запомнить легко: D-регистры предназначены для чисел double-точности, а S-регистры — для чисел single-точности.

Больше об этом: ?? (стр. ??).

Обе константы (3,14 и 4,1) хранятся в памяти в формате IEEE 754.

Инструкции VLDR и VMOV, как можно догадаться, это аналоги обычных LDR и MOV, но они работают с D-регистрами.

Важно отметить, что эти инструкции, как и D-регистры, предназначены не только для работы с числами с плавающей точкой, но пригодны также и для работы с SIMD (NEON), и позже это также будет видно.

Аргументы передаются в функцию обычным путем через R-регистры, однако каждое число, имеющее двойную точность, занимает 64 бита, так что для передачи каждого нужны два R-регистра.

VMOV D17, R0, R1 в самом начале составляет два 32-битных значения из R0 и R1 в одно 64-битное и сохраняет в D17.

VMOV R0, R1, D16 в конце это обратная процедура: то что было в D16 остается в двух регистрах R0 и R1, потому что число с двойной точностью, занимающее 64 бита, возвращается в паре регистров R0 и R1.

VDIV, VMUL и VADD, это инструкции для работы с числами с плавающей точкой, вычисляющие, соответственно, **частное**, **произведение** и сумму.

Код для Thumb-2 такой же.

### ARM: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
f
    PUSH    {R3-R7,LR}
    MOVS    R7, R2
    MOVS    R4, R3
    MOVS    R5, R0
    MOVS    R6, R1
    LDR     R2, =0x66666666 ; 4.1
    LDR     R3, =0x40106666
    MOVS    R0, R7
    MOVS    R1, R4
    BL     __aeabi_dmul
    MOVS    R7, R0
    MOVS    R4, R1
    LDR     R2, =0x51EB851F ; 3.14
    LDR     R3, =0x40091EB8
    MOVS    R0, R5
    MOVS    R1, R6
    BL     __aeabi_ddiv
    MOVS    R2, R7
    MOVS    R3, R4
    BL     __aeabi_dadd
    POP     {R3-R7,PC}

; 4.1 в формате IEEE 754:
dword_364    DCD 0x66666666 ; DATA XREF: f+A
dword_368    DCD 0x40106666 ; DATA XREF: f+C
; 3.14 в формате IEEE 754:
dword_36C    DCD 0x51EB851F ; DATA XREF: f+1A
dword_370    DCD 0x40091EB8 ; DATA XREF: f+1C
```

Keil компилировал для процессора, в котором может и не быть поддержки FPU или NEON. Так что числа с двойной точностью передаются в парах обычных R-регистров, а вместо FPU-инструкций вызываются сервисные библиотечные функции

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

`__aeabi_dmul`, `__aeabi_ddiv`, `__aeabi_dadd`, эмулирующие умножение, деление и сложение чисел с плавающей точкой.

Конечно, это медленнее чем FPU-сопроцессор, но это лучше, чем ничего.

Кстати, похожие библиотеки для эмуляции сопроцессорных инструкций были очень распространены в x86 когда сопроцессор был редким и дорогим и присутствовал далеко не во всех компьютерах.

Эмуляция FPU-сопроцессора в ARM называется *soft float* или *armel (emulation)*, а использование FPU-инструкций сопроцессора — *hard float* или *armhf*.

### ARM64: Оптимизирующий GCC (Linaro) 4.9

Очень компактный код:

Листинг 1.210: Оптимизирующий GCC (Linaro) 4.9

```
f:
; D0 = a, D1 = b
    ldr    d2, .LC25      ; 3.14
; D2 = 3.14
    fdiv  d0, d0, d2
; D0 = D0/D2 = a/3.14
    ldr    d2, .LC26      ; 4.1
; D2 = 4.1
    fmadd d0, d1, d2, d0
; D0 = D1*D2+D0 = b*4.1+a/3.14
    ret

; константы в формате IEEE 754:
.LC25:
    .word 1374389535      ; 3.14
    .word 1074339512
.LC26:
    .word 1717986918      ; 4.1
    .word 1074816614
```

### ARM64: Неоптимизирующий GCC (Linaro) 4.9

Листинг 1.211: Неоптимизирующий GCC (Linaro) 4.9

```
f:
    sub   sp, sp, #16
    str   d0, [sp,8]      ; сохранить "a" в Register Save Area
    str   d1, [sp]        ; сохранить "b" в Register Save Area
    ldr   x1, [sp,8]
; X1 = a
    ldr   x0, .LC25
; X0 = 3.14
    fmov  d0, x1
    fmov  d1, x0
; D0 = a, D1 = 3.14
    fdiv  d0, d0, d1
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; D0 = D0/D1 = a/3.14
    fmov    x1, d0
; X1 = a/3.14
    ldr    x2, [sp]
; X2 = b
    ldr    x0, .LC26
; X0 = 4.1
    fmov    d0, x2
; D0 = b
    fmov    d1, x0
; D1 = 4.1
    fmul   d0, d0, d1
; D0 = D0*D1 = b*4.1

    fmov    x0, d0
; X0 = D0 = b*4.1
    fmov    d0, x1
; D0 = a/3.14
    fmov    d1, x0
; D1 = X0 = b*4.1
    fadd   d0, d0, d1
; D0 = D0+D1 = a/3.14 + b*4.1

    fmov    x0, d0 ; \ избыточный код
    fmov    d0, x0 ; /
    add    sp, sp, 16
    ret
.LC25:
    .word   1374389535      ; 3.14
    .word   1074339512
.LC26:
    .word   1717986918      ; 4.1
    .word   1074816614

```

Неоптимизирующий GCC более многословный. Здесь много ненужных перетасовок значений, включая явно избыточный код (последние две инструкции GMOV). Должно быть, GCC 4.9 пока ещё не очень хорош для генерации кода под ARM64. Интересно заметить что у ARM64 64-битные регистры и D-регистры так же 64-битные. Так что компилятор может сохранять значения типа *double* в **GPR!** вместо локального стека. Это было невозможно на 32-битных CPU. И снова, как упражнение, вы можете попробовать соптимизировать эту функцию вручную, без добавления новых инструкций вроде FMADD.

### 1.25.6. Передача чисел с плавающей запятой в аргументах

```

#include <math.h>
#include <stdio.h>

int main ()
{
    printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    return 0;
}

```

**x86**

Посмотрим, что у нас вышло (MSVC 2010):

Листинг 1.212: MSVC 2010

```

CONST    SEGMENT
__real@40400147ae147ae1 DQ 040400147ae147ae1r    ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r    ; 1.54
CONST    ENDS

_main    PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 8 ; выделить место для первой переменной
    fld    QWORD PTR __real@3ff8a3d70a3d70a4
    fstp   QWORD PTR [esp]
    sub     esp, 8 ; выделить место для второй переменной
    fld    QWORD PTR __real@40400147ae147ae1
    fstp   QWORD PTR [esp]
    call   _pow
    add     esp, 8 ; вернуть место от одной переменной.

; в локальном стеке сейчас все еще зарезервировано 8 байт для нас.
; результат сейчас в ST(0)

; перегрузить результат из ST(0) в локальный стек для printf():
    fstp   QWORD PTR [esp]
    push   OFFSET $SG2651
    call   _printf
    add    esp, 12
    xor    eax, eax
    pop    ebp
    ret    0
_main    ENDP

```

FLD и FSTP перемещают переменные из сегмента данных в FPU-стек или обратно. `pow()`<sup>113</sup> достает оба значения из стека и возвращает результат в ST(0). `printf()` берет 8 байт из стека и трактует их как переменную типа *double*.

Кстати, с тем же успехом можно было бы перекладывать эти два числа из памяти в стек при помощи пары MOV:

ведь в памяти числа в формате IEEE 754, `pow()` также принимает их в том же формате, и никакая конверсия не требуется.

Собственно, так и происходит в следующем примере с ARM: ?? (стр. ??).

<sup>113</sup> стандартная функция Си, возводящая число в степень

**ARM + Неоптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)**

```

_main
var_C      = -0xC

        PUSH    {R7,LR}
        MOV     R7, SP
        SUB     SP, SP, #4
        VLDR   D16, =32.01
        VMOV   R0, R1, D16
        VLDR   D16, =1.54
        VMOV   R2, R3, D16
        BLX    _pow
        VMOV   D16, R0, R1
        MOV     R0, 0xFC1 ; "32.01 ^ 1.54 = %lf\n"
        ADD    R0, PC
        VMOV   R1, R2, D16
        BLX    _printf
        MOVS   R1, 0
        STR    R0, [SP,#0xC+var_C]
        MOV     R0, R1
        ADD    SP, SP, #4
        POP    {R7,PC}

dbl_2F90  DCFD 32.01      ; DATA XREF: _main+6
dbl_2F98  DCFD 1.54      ; DATA XREF: _main+E

```

Как уже было указано, 64-битные числа с плавающей точкой передаются в парах R-регистров.

Этот код слегка избыточен (наверное, потому что не включена оптимизация), ведь можно было бы загружать значения напрямую в R-регистры минуя загрузку в D-регистры.

Итак, видно, что функция `_pow` получает первый аргумент в R0 и R1, а второй в R2 и R3. Функция оставляет результат в R0 и R1. Результат работы `_pow` перекладывается в D16, затем в пару R1 и R2, откуда `printf()` берет это число-результат.

**ARM + Неоптимизирующий Keil 6/2013 (Режим ARM)**

```

_main
        STMFD   SP!, {R4-R6,LR}
        LDR     R2, =0xA3D70A4 ; y
        LDR     R3, =0x3FF8A3D7
        LDR     R0, =0xAE147AE1 ; x
        LDR     R1, =0x40400147
        BL     pow
        MOV     R4, R0
        MOV     R2, R4
        MOV     R3, R1
        ADR     R0, a32_011_54Lf ; "32.01 ^ 1.54 = %lf\n"

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

BL      __2printf
MOV     R0, #0
LDMFD  SP!, {R4-R6,PC}

y          DCD 0xA3D70A4          ; DATA XREF: _main+4
dword_520  DCD 0x3FF8A3D7        ; DATA XREF: _main+8
x          DCD 0xAE147AE1        ; DATA XREF: _main+C
dword_528  DCD 0x40400147        ; DATA XREF: _main+10
a32_011_54Lf  DCB "32.01 ^ 1.54 = %lf",0xA,0
                                           ; DATA XREF: _main+24

```

Здесь не используются D-регистры, используются только пары R-регистров.

## ARM64 + Оптимизирующий GCC (Linaro) 4.9

Листинг 1.213: Оптимизирующий GCC (Linaro) 4.9

```

f:
    stp    x29, x30, [sp, -16]!
    add    x29, sp, 0
    ldr    d1, .LC1 ; загрузить 1.54 в D1
    ldr    d0, .LC0 ; загрузить 32.01 в D0
    bl     pow
; результат pow() в D0
    adrp   x0, .LC2
    add    x0, x0, :lo12:.LC2
    bl     printf
    mov    w0, 0
    ldp    x29, x30, [sp], 16
    ret

.LC0:
; 32.01 в формате IEEE 754
    .word  -1374389535
    .word  1077936455

.LC1:
; 1.54 в формате IEEE 754
    .word  171798692
    .word  1073259479

.LC2:
    .string "32.01 ^ 1.54 = %lf\n"

```

Константы загружаются в D0 и D1: функция `pow()` берет их оттуда. Результат в D0 после исполнения `pow()`. Он пропускается в `printf()` без всякой модификации и перемещений, потому что `printf()` берет аргументы [интегральных типов](#) и указатели из X-регистров, а аргументы типа плавающей точки из D-регистров.

### 1.25.7. Пример со сравнением

Попробуем теперь вот это:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

#include <stdio.h>

double d_max (double a, double b)
{
    if (a>b)
        return a;

    return b;
};

int main()
{
    printf ("%f\n", d_max (1.2, 3.4));
    printf ("%f\n", d_max (5.6, -4));
};

```

Несмотря на кажущуюся простоту этой функции, понять, как она работает, будет чуть сложнее.

## x86

### Неоптимизирующий MSVC

Вот что выдал MSVC 2010:

Листинг 1.214: Неоптимизирующий MSVC 2010

```

PUBLIC    _d_max
_TEXT    SEGMENT
_a$ = 8          ; size = 8
_b$ = 16        ; size = 8
_d_max   PROC
    push   ebp
    mov    ebp, esp
    fld   QWORD PTR _b$[ebp]

; состояние стека сейчас: ST(0) = _b
; сравниваем _b (в ST(0)) и _a, затем выталкиваем значение из стека

    fcomp  QWORD PTR _a$[ebp]

; стек теперь пустой

    fnstsw ax
    test  ah, 5
    jp    SHORT $LN1@d_max

; мы здесь только если a>b

    fld   QWORD PTR _a$[ebp]
    jmp   SHORT $LN2@d_max
$LN1@d_max:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

fld    QWORD PTR _b$[ebp]
$LN2@d_max:
pop    ebp
ret    0
_d_max    ENDP

```

Итак, FLD загружает `_b` в регистр `ST(0)`.

FCOMP сравнивает содержимое `ST(0)` с тем, что лежит в `_a` и выставляет биты `C3/C2/C0` в регистре статуса FPU. Это 16-битный регистр отражающий текущее состояние FPU.

После этого инструкция FCOMP также выдергивает одно значение из стека. Это отличает её от FCOM, которая просто сравнивает значения, оставляя стек в таком же состоянии.

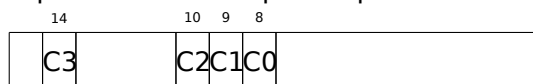
К сожалению, у процессоров до Intel P6<sup>114</sup> нет инструкций условного перехода, проверяющих биты `C3/C2/C0`. Возможно, так сложилось исторически (вспомните о том, что FPU когда-то был вообще отдельным чипом).

А у Intel P6 появились инструкции FCOMI/FCOMIP/FUCOMI/FUCOMIP, делающие то же самое, только напрямую модифицирующие флаги `ZF/PF/CF`.

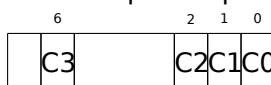
Так что FNSTSW копирует содержимое регистра статуса в `AX`. Биты `C3/C2/C0` занимают позиции, соответственно, 14, 10, 8. В этих позициях они и остаются в регистре `AX`, и все они расположены в старшей части регистра — `AH`.

- Если  $b > a$  в нашем случае, то биты `C3/C2/C0` должны быть выставлены так: 0, 0, 0.
- Если  $a > b$ , то биты будут выставлены: 0, 0, 1.
- Если  $a = b$ , то биты будут выставлены так: 1, 0, 0.
- Если результат не определен (в случае ошибки), то биты будут выставлены так: 1, 1, 1.

Вот как биты `C3/C2/C0` расположены в регистре `AX`:



Вот как биты `C3/C2/C0` расположены в регистре `AH`:



После исполнения `test ah, 5`<sup>115</sup> будут учтены только биты `C0` и `C2` (на позициях 0 и 2), остальные просто проигнорированы.

Теперь немного о *parity flag*<sup>116</sup>. Ещё один замечательный рудимент эпохи.

Этот флаг выставляется в 1 если количество единиц в последнем результате четно. И в 0 если нечетно.

<sup>114</sup>Intel P6 это Pentium Pro, Pentium II, и последующие модели

<sup>115</sup>5=101b

<sup>116</sup>флаг четности

Заглянем в Wikipedia<sup>117</sup>:

One common reason to test the parity flag actually has nothing to do with parity. The FPU has four condition flags (C0 to C3), but they cannot be tested directly, and must instead be first copied to the flags register. When this happens, C0 is placed in the carry flag, C2 in the parity flag and C3 in the zero flag. The C2 flag is set when e.g. incomparable floating point values (NaN or unsupported format) are compared with the FUCOM instructions.

Как упоминается в Wikipedia, флаг четности иногда используется в FPU-коде и сейчас мы увидим как.

Флаг PF будет выставлен в 1, если C0 и C2 оба 1 или оба 0. И тогда сработает последующий JP (*jump if PF==1*). Если мы вернемся чуть назад и посмотрим значения C3/C2/C0 для разных вариантов, то увидим, что условный переход JP сработает в двух случаях: если  $b > a$  или если  $a = b$  (ведь бит C3 перестал учитываться после исполнения `test ah, 5`).

Дальше всё просто. Если условный переход сработал, то FLD загрузит значение `_b` в `ST(0)`, а если не сработал, то загрузится `_a` и произойдет выход из функции.

### А как же проверка флага C2?

Флаг C2 включается в случае ошибки (NaN, и т. д.), но наш код его не проверяет.

Если программисту нужно знать, не произошла ли FPU-ошибка, он должен позаботиться об этом дополнительно, добавив соответствующие проверки.

---

<sup>117</sup>[https://en.wikipedia.org/wiki/Parity\\_flag](https://en.wikipedia.org/wiki/Parity_flag)

## Первый пример с OllyDbg: $a=1,2$ и $b=3,4$

Загружаем пример в OllyDbg:

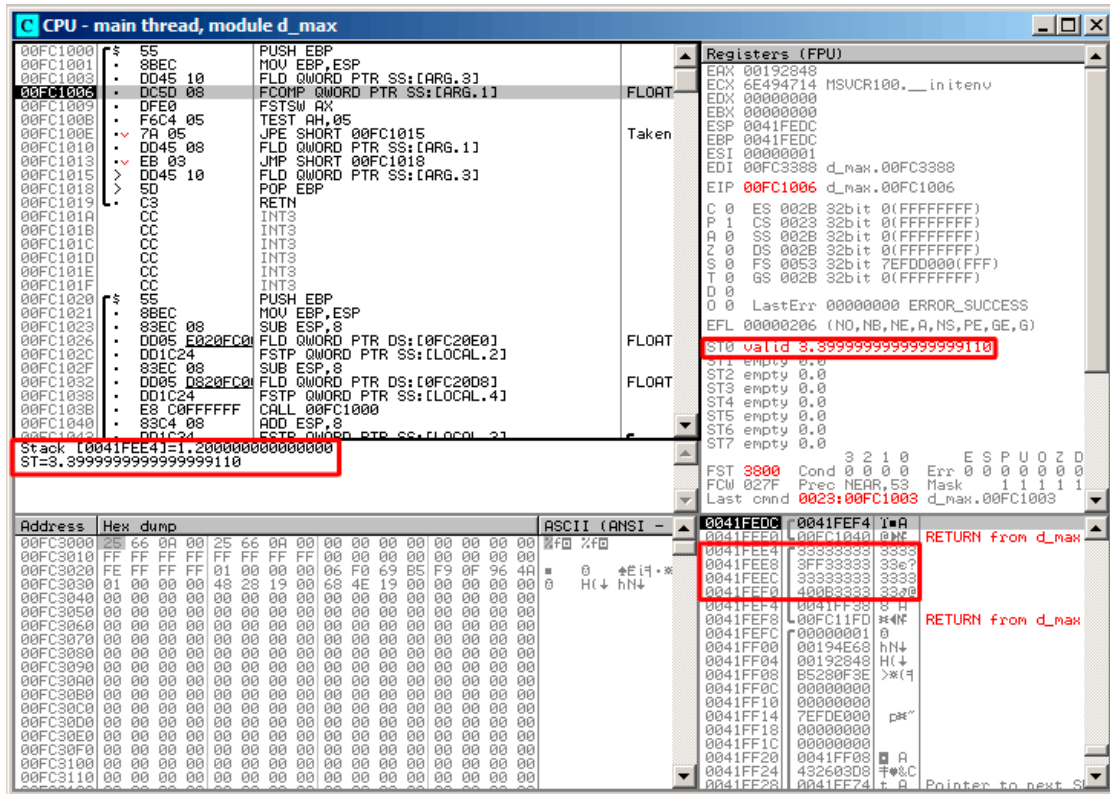


Рис. 1.67: OllyDbg: первая FLD исполнилась

Текущие параметры функции:  $a = 1,2$  и  $b = 3,4$  (их видно в стеке: 2 пары 32-битных значений).  $b$  (3,4) уже загружено в ST(0). Сейчас будет исполняться FCOMP. OllyDbg показывает второй аргумент для FCOMP, который сейчас находится в стеке.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

FCOMP отработал:

The screenshot shows the OlyDbg interface with the following details:

- CPU - main thread, module d\_max:**
  - Address 00FC1000: 55 PUSH EBP
  - Address 00FC1001: 8BEC MOV EBP, ESP
  - Address 00FC1003: DD45 10 FLD QWORD PTR SS:[ARG.3]
  - Address 00FC1006: DC5D 08 FCOMP QWORD PTR SS:[ARG.1]
  - Address 00FC1009: DFE0 FSTSW AX
  - Address 00FC100B: F6C4 05 TEST AH, 05
  - Address 00FC100E: 7A 05 JPE SHORT 00FC1015
  - Address 00FC1010: DD45 08 FLD QWORD PTR SS:[ARG.1]
  - Address 00FC1013: EB 03 JMP SHORT 00FC1018
  - Address 00FC1015: DD45 10 FLD QWORD PTR SS:[ARG.3]
  - Address 00FC1018: 5D POP EBP
  - Address 00FC1019: C3 RETN
  - Address 00FC101A: CC INT3
  - Address 00FC101B: CC INT3
  - Address 00FC101C: CC INT3
  - Address 00FC101D: CC INT3
  - Address 00FC101E: CC INT3
  - Address 00FC101F: CC INT3
  - Address 00FC1020: 55 PUSH EBP
  - Address 00FC1021: 8BEC MOV EBP, ESP
  - Address 00FC1023: 83EC 08 SUB ESP, 8
  - Address 00FC1026: DD05 0820FC01 FLD QWORD PTR DS:[0FC20E01]
  - Address 00FC1029: DD1C24 FSTP QWORD PTR SS:[LOCAL.2]
  - Address 00FC102F: 83EC 08 SUB ESP, 8
  - Address 00FC1032: DD05 0820FC01 FLD QWORD PTR DS:[0FC20E01]
  - Address 00FC1035: DD1C24 FSTP QWORD PTR SS:[LOCAL.4]
  - Address 00FC1038: E8 C0FFFFFF CALL 00FC1000
  - Address 00FC1040: 83C4 08 ADD ESP, 8
  - Address 00FC1043: DD1C24 FSTP QWORD PTR SS:[LOCAL.2]
- Registers (FPU):**
  - EAX: 00192848
  - ECX: 6E494714 MSUCR100.\_\_initenv
  - EDX: 00000000
  - EBX: 00000000
  - ESP: 0041FEDC
  - EBP: 0041FEDC
  - ESI: 00000001
  - EDI: 00FC3388 d\_max.00FC3388
  - EIP: 00FC1009 d\_max.00FC1009
  - D 0
  - I 0
  - O 0 LastErr: 00000000 ERROR\_SUCCESS
  - EFL: 00000206 (NO, NB, NE, A, NS, PE, GE, G)
  - ST0 empty 0.0
  - ST1 empty 0.0
  - ST2 empty 0.0
  - ST3 empty 0.0
  - ST4 empty 0.0
  - ST5 empty 0.0
  - ST6 empty 0.0
  - ST7 empty 0.0
  - Condition Code Register (FST): 0000 (Cond: 0 0 0 0) Err: 0 0 0 0 0 0
  - FCW: 027F (Prec: NORM, SS) Mask: 1 1 1 1
  - Last cmd: 0023:00FC1006 d\_max.00FC1006
- Address Hex dump:**
  - 00FC3000: 25 66 0A 00 25 66 0A 00 00 00 00 00 00 00 00 00
  - 00FC3010: FF FF FF FF FF FF FF FF 00 00 00 00 00 00 00 00
  - 00FC3020: FE FF FF FF 01 00 00 00 06 F0 69 B5 F9 0F 96 4A
  - 00FC3030: 01 00 00 00 48 28 19 00 68 4E 19 00 00 00 00 00
  - 00FC3040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - 00FC3050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - 00FC3060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - 00FC3070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - 00FC3080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - 00FC3090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - 00FC30A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - 00FC30B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - 00FC30C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - 00FC30D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - 00FC30E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - 00FC30F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - 00FC3100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  - 00FC3110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
- Stack (Address Hex dump):**
  - 0041FEFC: 00FC1040 00FC1040 RETURN from d\_max
  - 0041FEE0: 33333333 3333
  - 0041FEE4: 33333333 3333
  - 0041FEF0: 33333333 3333
  - 0041FEF4: 400B3333 3333
  - 0041FEF8: 0041FF38 0A
  - 0041FEFC: 00FC11F0 00FC11F0 RETURN from d\_max
  - 0041FE00: 00000001 0
  - 0041FF00: 00194E68 hN
  - 0041FF04: 00192948 hN
  - 0041FF08: 85280F3E >\*(q
  - 0041FF0C: 00000000
  - 0041FF10: 00000000
  - 0041FF14: 7EFD0E00 p&#
  - 0041FF18: 00000000
  - 0041FF1C: 00000000
  - 0041FF20: 0041FF08 A
  - 0041FF24: 43260308 \*%C
  - 0041FF28: 0041FE74 t, a Printer to next S...

Рис. 1.68: OlyDbg: FCOMP исполнилась

Мы видим состояния condition-флагов **FPU!**: все нули. Вытолкнутое значение отображается как ST(7). Почему это так, объяснялось ранее: ?? (стр. ??).

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

FNSTSW сработал:

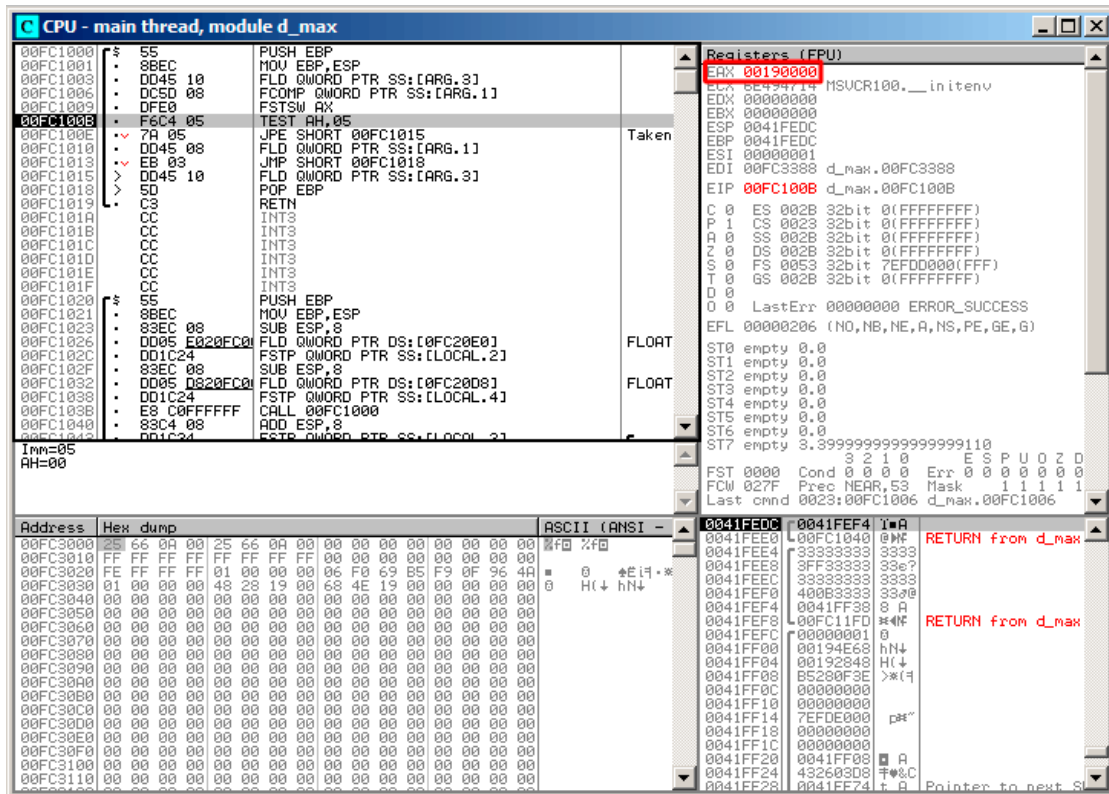


Рис. 1.69: OllyDbg: FNSTSW исполнялась

Видно, что регистр AX содержит нули. Действительно, ведь все condition-флаги тоже содержали нули.

(OllyDbg дизассемблирует команду FNSTSW как FSTSW — это синоним).

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

TEST сработал:

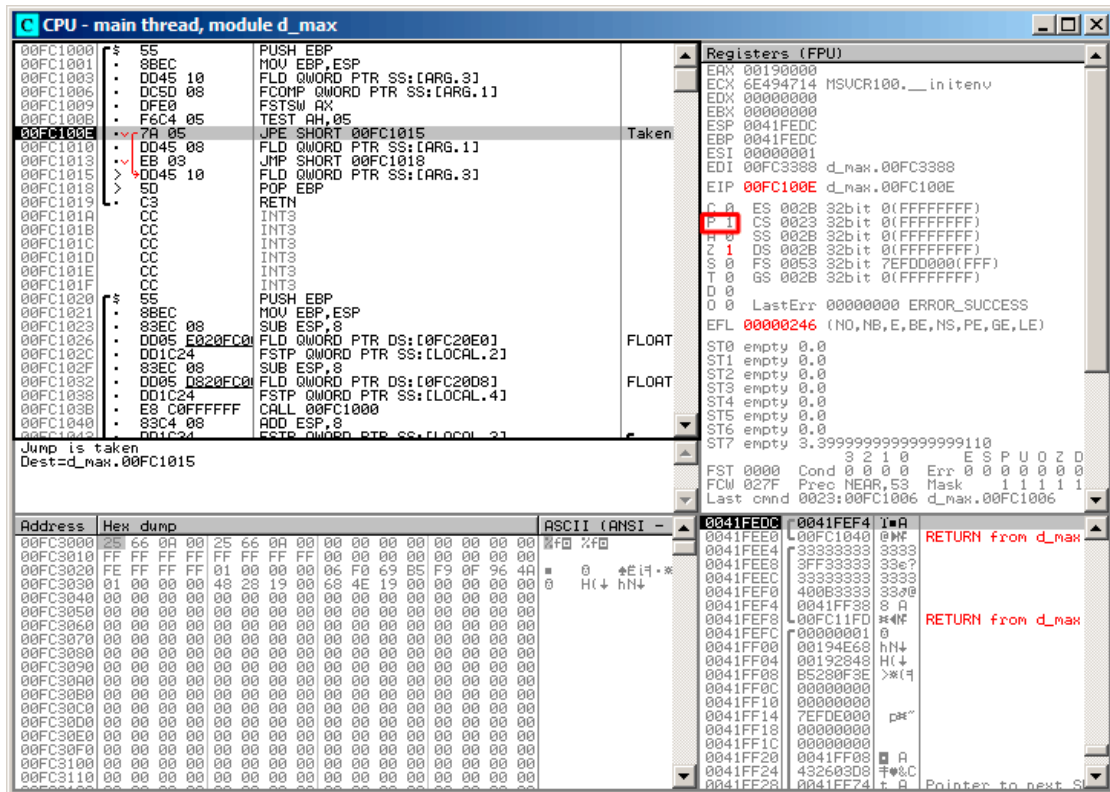


Рис. 1.70: OllyDbg: TEST исполнилась

Флаг PF равен единице. Всё верно: количество выставленных бит в 0 — это 0, а 0 — это четное число.

OllyDbg дизассемблирует JP как **JPE!**<sup>118</sup> — это синонимы. И она сейчас работает.

<sup>118</sup> **JPE!**

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

**JPE!** работала, FLD загрузила в ST(0) значение  $b(3,4)$ :

The screenshot displays the OllyDbg interface for the CPU window, showing the disassembly of instructions. The instruction at address 00FC1019 is RETN. The Registers (FPU) window shows the status of the floating-point registers, with ST0 containing the value 3.3999999999999999110. The stack window shows the return address 0041FEE0.

Address	Hex dump	ASCII (ANSI)	Disassembly
00FC1000	55		PUSH EBP
00FC1001	8BEC		MOV EBP,ESP
00FC1003	DD45 10		FLD QWORD PTR SS:[ARG.3]
00FC1006	DC5D 08		FCOMP QWORD PTR SS:[ARG.1]
00FC1009	DFE0		FSTSW AX
00FC100B	F6C4 05		TEST AH,05
00FC100E	7A 05		JPE SHORT 00FC1015
00FC1010	DD45 08		FLD QWORD PTR SS:[ARG.1]
00FC1013	EB 03		JMP SHORT 00FC1018
00FC1015	DD45 10		FLD QWORD PTR SS:[ARG.3]
00FC1018	5D		POP EBP
00FC1019	C3		RETN

Рис. 1.71: OllyDbg: вторая FLD исполнилась

Функция заканчивает свою работу.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



## Второй пример с OllyDbg: $a=5,6$ и $b=-4$

Загружаем пример в OllyDbg:

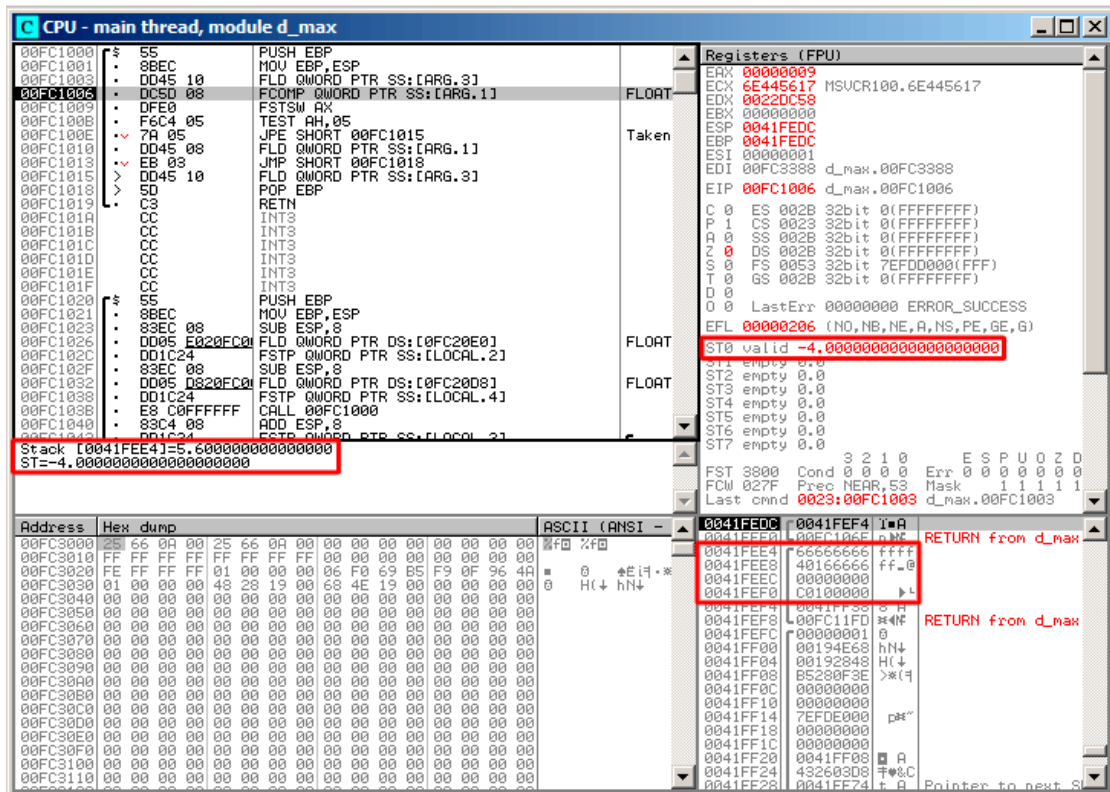


Рис. 1.72: OllyDbg: первая FLD исполнилась

Текущие параметры функции:  $a = 5,6$  и  $b = -4$ .  $b$  (-4) уже загружено в ST(0). Сейчас будет исполняться FCOMP. OllyDbg показывает второй аргумент FCOMP, который сейчас находится в стеке.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



FCOMP отработал:

The screenshot shows the OllDbg interface with the following details:

- CPU - main thread, module d\_max:**
  - Address 00FC1009: `F6C4 05` TEST AH, 05 (Taken)
  - Address 00FC100E: `7A 05` JPE SHORT 00FC1015 (Taken)
  - Address 00FC1010: `DD45 08` FLD QWORD PTR SS:[ARG.1] (Taken)
  - Address 00FC1013: `EB 03` JMP SHORT 00FC1018 (Taken)
  - Address 00FC1015: `DD45 10` FLD QWORD PTR SS:[ARG.3] (Taken)
  - Address 00FC1018: `5D` POP EBP (Taken)
  - Address 00FC1019: `C3` RETN (Taken)
  - Address 00FC1020: `55` PUSH EBP (Taken)
  - Address 00FC1021: `8BEC` MOV EBP, ESP (Taken)
  - Address 00FC1023: `83EC 08` SUB ESP, 8 (Taken)
  - Address 00FC1026: `DD05 F020FC0` FLD QWORD PTR DS:[0FC20E0] (Taken)
  - Address 00FC102C: `DD1C24` FSTP QWORD PTR SS:[LOCAL.2] (Taken)
  - Address 00FC102F: `83EC 08` SUB ESP, 8 (Taken)
  - Address 00FC1032: `DD05 D820FC0` FLD QWORD PTR DS:[0FC20D8] (Taken)
  - Address 00FC1038: `DD1C24` FSTP QWORD PTR SS:[LOCAL.4] (Taken)
  - Address 00FC103B: `E8 C0FFFFFF` CALL 00FC1000 (Taken)
  - Address 00FC1040: `83C4 08` ADD ESP, 8 (Taken)
  - Address 00FC1043: `DD1C24` FSTP QWORD PTR SS:[LOCAL.2] (Taken)
- Registers (FPU):**
  - ST0 empty 0.0
  - ST1 empty 0.0
  - ST2 empty 0.0
  - ST3 empty 0.0
  - ST4 empty 0.0
  - ST5 empty 0.0
  - ST6 empty 0.0
  - ST7 empty 0.0
  - Control flags: `3 2 1 0` (highlighted in red)
  - Err: 0 0 0 0 0 0
  - Mask: 1 1 1 1
- Condition Code:** `FST 0100 Cond 0 0 0 1` (highlighted in red)
- Memory Dump:** Shows hex dump and ASCII (ANSI) for addresses 00FC3000 to 00FC3110.
- Stack:** Shows return addresses like `0041FEF4` and `0041FEF8` with labels `RETURN from d_max`.

Рис. 1.73: OllDbg: FCOMP исполнилась

Мы видим значения condition-флагов **FPU!**: все нули, кроме C0.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

FNSTSW сработал:

The screenshot shows the OllyDbg interface with the CPU window displaying assembly code for the 'main thread, module d\_max'. The instruction at address 00FC100B is highlighted: `F6C4 05 TEST AH,05`, which is marked as 'Taken'. The 'Registers (FPU)' window shows the `AX` register containing `00000100`. The 'LastErr' field in the status bar shows `ERROR_SUCCESS`. The 'Registers (CPU)' window shows the `AX` register with the value `00000100`. The 'Registers (FPU)' window shows the `AX` register with the value `00000100`. The 'Registers (CPU)' window shows the `AX` register with the value `00000100`. The 'Registers (FPU)' window shows the `AX` register with the value `00000100`.

Address	Hex dump	ASCII (ANSI)	Registers (CPU)	Registers (FPU)
00FC1000	55		AX 00000100	AX 00000100
00FC1001	8BEC		EDX 0022DC53	EDX 0022DC53
00FC1003	DD45 10		EBX 00000000	EBX 00000000
00FC1006	DC5D 08		ESP 0041FEDC	ESP 0041FEDC
00FC1009	DFF0		EBP 0041FEDC	EBP 0041FEDC
00FC100B	F6C4 05		ESI 00000001	ESI 00000001
00FC100E	7A 05		EDI 00FC3388	EDI 00FC3388
00FC1010	DD45 08		EIP 00FC100B	EIP 00FC100B
00FC1013	EB 03		C 0 ES 002B 32bit 0(FFFFFFFF)	C 0 ES 002B 32bit 0(FFFFFFFF)
00FC1015	DD45 10		P 1 CS 0023 32bit 0(FFFFFFFF)	P 1 CS 0023 32bit 0(FFFFFFFF)
00FC1018	5D		A 0 SS 002B 32bit 0(FFFFFFFF)	A 0 SS 002B 32bit 0(FFFFFFFF)
00FC1019	C3		Z 0 DS 002B 32bit 0(FFFFFFFF)	Z 0 DS 002B 32bit 0(FFFFFFFF)
00FC101A	CC		S 0 FS 0053 32bit 7EFD0000(FFF)	S 0 FS 0053 32bit 7EFD0000(FFF)
00FC101B	CC		T 0 GS 002B 32bit 0(FFFFFFFF)	T 0 GS 002B 32bit 0(FFFFFFFF)
00FC101C	CC		D 0 LastErr 00000000 ERROR_SUCCESS	D 0 LastErr 00000000 ERROR_SUCCESS
00FC101D	CC		EFL 00000206 (NO,NB,NE,A,NS,PE,GE,G)	EFL 00000206 (NO,NB,NE,A,NS,PE,GE,G)
00FC101E	55		ST0 empty 0.0	ST0 empty 0.0
00FC1020	8BEC		ST1 empty 0.0	ST1 empty 0.0
00FC1021	83EC 08		ST2 empty 0.0	ST2 empty 0.0
00FC1023	DD1C24		ST4 empty 0.0	ST4 empty 0.0
00FC1026	DD1C24		ST5 empty 0.0	ST5 empty 0.0
00FC102C	DD1C24		ST6 empty 0.0	ST6 empty 0.0
00FC102F	83EC 08		ST7 empty -4.00000000000000000000	ST7 empty -4.00000000000000000000
00FC1032	DD05 0820FC04		FST 0100 Cond 0 0 0 1 Err 0 0 0 0 0 0	FST 0100 Cond 0 0 0 1 Err 0 0 0 0 0 0
00FC1033	DD1C24		FCW 027F Prec NEAR,53 Mask 1 1 1 1	FCW 027F Prec NEAR,53 Mask 1 1 1 1
00FC1038	E8 C0FFFFFF		Last cmd 0023:00FC1006 d_max.00FC1006	Last cmd 0023:00FC1006 d_max.00FC1006
00FC1038	DD1C24			
00FC1040	83C4 08			
00FC1043	DD1C24			

Рис. 1.74: OllyDbg: FNSTSW исполнялась

Видно, что регистр AX содержит 0x100: флаг C0 стал на место 8-го бита.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

TEST сработал:

The screenshot shows the OllyDbg interface with the following details:

- Assembly Window:** Address 00FC100E, Instruction: `TEST AH, 05`. Status: Taken.
- Registers (FPU) Window:** PF flag is 0. Other registers like EAX, ECX, EDI, etc., contain various values.
- Hex Dump Window:** Shows the instruction bytes: `25 66 0A 00 25 66 0A 00 00 00 00 00 00 00 00 00`.
- Disassembly Window:** Shows the instruction `TEST AH, 05` with a status of 'Taken'.

Рис. 1.75: OllyDbg: TEST исполнилась

Флаг PF равен нулю. Всё верно: количество единичных бит в 0x100 — 1, а 1 — нечетное число.

**JPE!** сейчас не сработает.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

JPE! не сработала, FLD загрузила в ST(0) значение *a* (5,6):

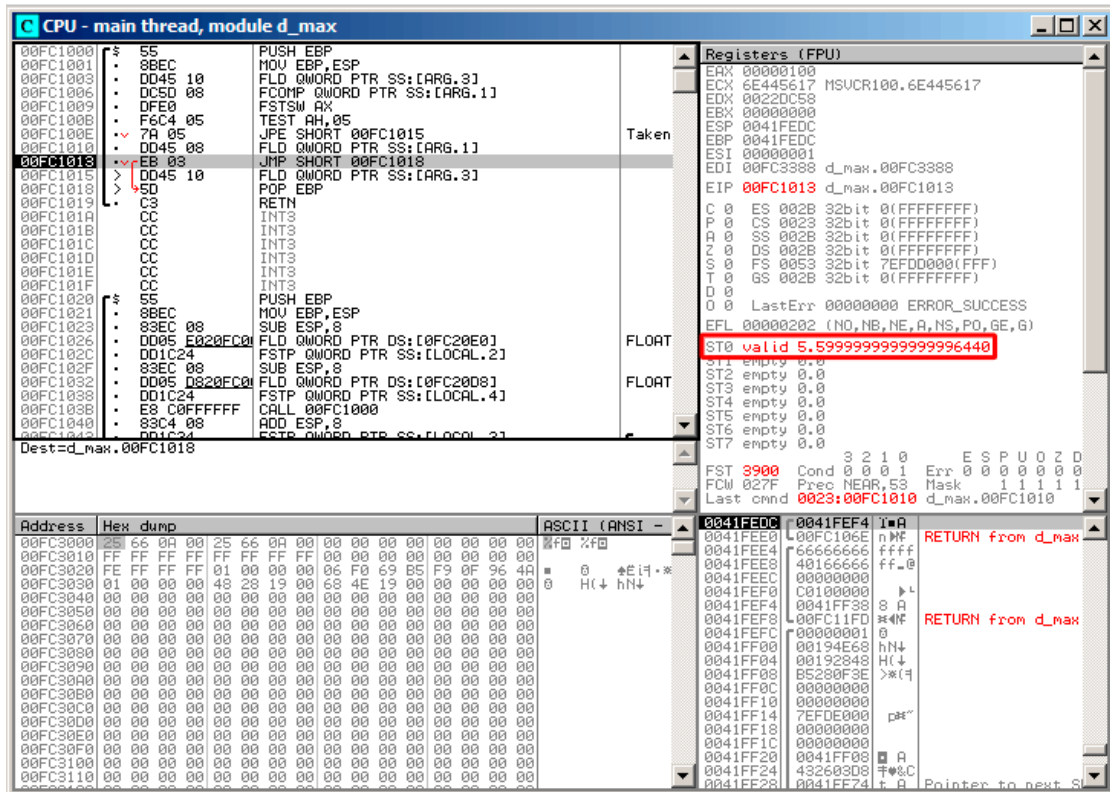


Рис. 1.76: OllyDbg: вторая FLD исполнилась

Функция заканчивает свою работу.

## Оптимизирующий MSVC 2010

Листинг 1.215: Оптимизирующий MSVC 2010

```

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_d_max PROC
    fld QWORD PTR _b$[esp-4]
    fld QWORD PTR _a$[esp-4]

; состояние стека сейчас: ST(0) = _a, ST(1) = _b

    fcom ST(1) ; сравнить _a и ST(1) = (_b)
    fnstsw ax
    test ah, 65 ; 00000041H
    jne SHORT $LN5@d_max

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; копировать содержимое ST(0) в ST(1) и вытолкнуть значение из стека,
; оставив _a на вершине
    fstp    ST(1)

; состояние стека сейчас: ST(0) = _a

    ret    0
$LN5@d_max:
; копировать содержимое ST(0) в ST(0) и вытолкнуть значение из стека,
; оставив _b на вершине
    fstp    ST(0)

; состояние стека сейчас: ST(0) = _b

    ret    0
_d_max    ENDP

```

FCOM отличается от FCOMP тем, что просто сравнивает значения и оставляет стек в том же состоянии. В отличие от предыдущего примера, операнды здесь в обратном порядке. Поэтому и результат сравнения в C3/C2/C0 будет отличаться:

- Если  $a > b$ , то биты C3/C2/C0 должны быть выставлены так: 0, 0, 0.
- Если  $b > a$ , то биты будут выставлены так: 0, 0, 1.
- Если  $a = b$ , то биты будут выставлены так: 1, 0, 0.

Инструкция `test ah, 65` как бы оставляет только два бита — C3 и C0. Они оба будут нулями, если  $a > b$ : в таком случае переход JNE не сработает. Далее имеется инструкция `FSTP ST(1)` — эта инструкция копирует значение ST(0) в указанный операнд и выдергивает одно значение из стека. В данном случае, она копирует ST(0) (где сейчас лежит `_a`) в ST(1). После этого на вершине стека два раза лежит `_a`. Затем одно значение выдергивается. После этого в ST(0) остается `_a` и функция завершается.

Условный переход JNE сработает в двух других случаях: если  $b > a$  или  $a = b$ . ST(0) скопируется в ST(0) (как бы холостая операция). Затем одно значение из стека вылетит и на вершине стека останется то, что до этого лежало в ST(1) (то есть `_b`). И функция завершится. Эта инструкция используется здесь видимо потому что в FPU нет другой инструкции, которая просто выдергивает значение из стека и выбрасывает его.

## Первый пример с OllyDbg: a=1,2 и b=3,4

Обе FLD отработали:

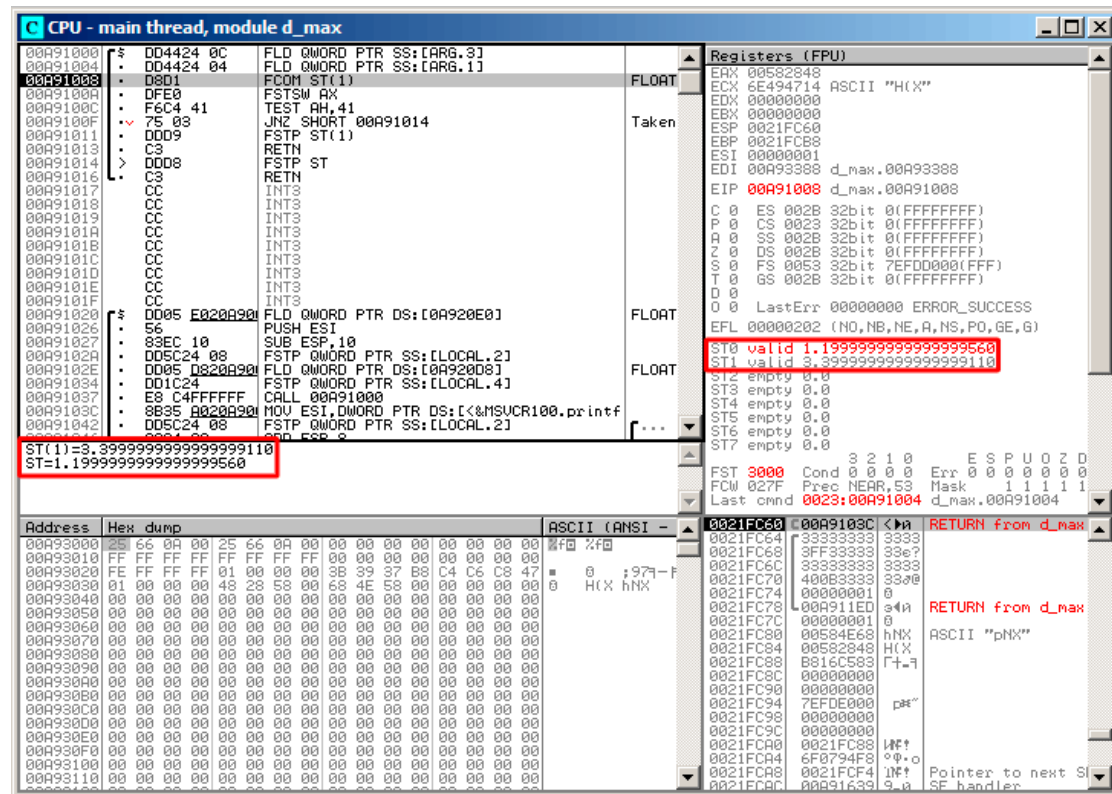


Рис. 1.77: OllyDbg: обе FLD исполнились

Сейчас будет исполняться FCOM: OllyDbg показывает содержимое ST(0) и ST(1) для удобства.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

FCOM работала:

```

CPU - main thread, module d_max
00A91000  |$ DD4424 0C  | FLD QWORD PTR SS:[ARG.3]
00A91004  | DD4424 04  | FLD QWORD PTR SS:[ARG.1]
00A91008  | D8D1      | FCOM ST(1)
00A9100A  | DFE0     | FSTSW AX
00A9100C  | F6C4 41  | TEST AH,41
00A9100F  | 75 03    | JNZ SHORT 00A91014
00A91011  | DDD9     | FSTP ST(1)
00A91013  | C3      | RETH
00A91014  | DDD8     | FSTP ST
00A91016  | C3      | RETH
00A91017  | CC      | INT3
00A91018  | CC      | INT3
00A91019  | CC      | INT3
00A9101A  | CC      | INT3
00A9101B  | CC      | INT3
00A9101C  | CC      | INT3
00A9101D  | CC      | INT3
00A9101E  | CC      | INT3
00A9101F  | CC      | INT3
00A91020  | DD05 F020A901 | FLD QWORD PTR DS:[0A920E0]
00A91026  | 56      | PUSH ESI
00A91027  | 83EC 10 | SUB ESP,10
00A9102A  | DD5C24 08 | FSTP QWORD PTR SS:[LOCAL.2]
00A9102E  | DD05 D820A901 | FLD QWORD PTR DS:[0A92008]
00A91034  | DD1C24   | FSTP QWORD PTR SS:[LOCAL.4]
00A91037  | E8 C4FFFFFF | CALL 00A91000
00A9103C  | 8B35 0020A901 | MOV ESI,DWORD PTR DS:[<&MSUCR100.printf]
00A91042  | DD5C24 08 | FSTP QWORD PTR SS:[LOCAL.2]
00A91044  | 9924 00  | FST QWORD PTR SS:[LOCAL.2]

FST=3100 (C3=0 C2=0 C1=0 C0=1 ES=0 SF=0 PE=0 UE=0 OE=0 ZE=0 DE=0 IE=0)
AX=2848

Registers (FPU)
EAX 00582848
ECX 6E494714 ASCII "H(X)"
EDX 00000000
EBX 00000000
ESP 0021FC60
EBP 0021FCB8
ESI 00000001
EDI 00A93388 d_max.00A93388
EIP 00A9100A d_max.00A9100A

C 0 ES 002B 32bit 0(FFFFFFFF)
P 0 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 0 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 7EFD0000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr 00000000 ERROR_SUCCESS
EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)

ST0 valid 1.1999999999999999560
ST1 valid 3.3999999999999999110
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0

FST 3100 Cond 0 0 0 1 Err 0 0 0 0 0 0
FCW 027F Mask 1 1 1 1
Last cmd 0023:00A91008 d_max.00A91008

Address Hex dump ASCII (ANSI)
00A93000 25 66 0A 00 25 66 0A 00 00 00 00 00 00 00 00 00 %f0 %f0
00A93010 FF FF FF FF FF FF FF FF 00 00 00 00 00 00 00 00
00A93020 FE FF FF FF 01 00 00 00 3B 39 37 B8 C4 C6 C8 47
00A93030 01 00 00 00 48 28 58 00 68 4E 58 00 00 00 00 00
00A93040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A93050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A93060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A93070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A93080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A93090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A930A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A930B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A930C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A930D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A930E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A930F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A93100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A93110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

0021FC64 33333333 3333
0021FC68 3FF33333 33e?
0021FC6C 33333333 3333
0021FC70 400E3333 33e0
0021FC74 00000001 0
0021FC78 00A911ED e4i
0021FC7C 00000001 0
0021FC80 00584E68 hNX
0021FC84 00582848 hNX
0021FC88 0B16C593 ГГ=Г
0021FC8C 00000000
0021FC90 00000000
0021FC94 7EFD0E00 pH"
0021FC98 00000000
0021FC9C 00000000
0021FCA0 0021FC88 hX!
0021FCA4 6F0794F6 0*o
0021FCA8 0021FCF4 hX! Pointer to next St
0021FCAC 00A91629 9.u SF handler
  
```

Рис. 1.78: OllyDbg: FCOM исполняется

C0 установлен, остальные флаги сброшены.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



FNSTSW работала, AX=0x3100:

The screenshot shows the OllyDbg interface with the CPU window displaying assembly code and the Registers (FPU) window showing the state of the processor registers.

**Assembly Code (CPU window):**

```

00A91000  DD4424 0C  FLD QWORD PTR SS:[ARG.3]
00A91004  DD4424 04  FLD QWORD PTR SS:[ARG.1]
00A91008  0801      FCOM ST(1)
00A9100A  DFE0      FSTSW AX
00A9100C  F6C4 41   TEST AH,41
00A9100F  75 03     JNZ SHORT 00A91014
00A91011  DD09      FSTP ST(1)
00A91013  C3        RETN
00A91014  DD08      FSTP ST
00A91016  C3        RETN
00A91017  CC        INT3
00A91018  CC        INT3
00A91019  CC        INT3
00A9101A  CC        INT3
00A9101B  CC        INT3
00A9101C  CC        INT3
00A9101D  CC        INT3
00A9101E  CC        INT3
00A9101F  CC        INT3
00A91020  DD05 F020A900 FLD QWORD PTR DS:[0A920E0]
00A91026  56        PUSH ESI
00A91027  83EC 10   SUB ESP,10
00A9102A  DD5C24 08  FSTP QWORD PTR SS:[LOCAL.2]
00A9102E  DD05 0820A900 FLD QWORD PTR DS:[0A920D8]
00A91034  DD1C24    FSTP QWORD PTR SS:[LOCAL.4]
00A91037  E8 C4FFFF CALL 00A91000
00A9103C  8B35 0020A900 MOV ESI,DWORD PTR DS:[<&MSUCR100.pr intf
00A91042  DD5C24 08  FSTP QWORD PTR SS:[LOCAL.2]
00A91044  83C4 02   SUB ESP,2
  
```

**Registers (FPU) window:**

- EAX: 00583100 (ASCII "Administrator")
- ECX: 00747117 (ASCII "H(X")
- EDX: 00000000 (ASCII "H(X")
- EBX: 00000000
- ESP: 0021FC60
- EBP: 0021FC88
- ESI: 00000001
- EDI: 00A93388 d\_max.00A93388
- EIP: 00A9100C d\_max.00A9100C

**Registers (General Purpose) window:**

- C 0 ES 002B 32bit 0(FFFFFFFF)
- P 0 CS 0023 32bit 0(FFFFFFFF)
- A 0 SS 002B 32bit 0(FFFFFFFF)
- Z 0 DS 002B 32bit 0(FFFFFFFF)
- S 0 FS 0053 32bit 7EFD0000(FFF)
- T 0 GS 002B 32bit 0(FFFFFFFF)
- D 0
- O 0 LastErr 00000000 ERROR\_SUCCESS
- EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)
- ST0 valid 1.1999999999999999560
- ST1 valid 3.3999999999999999110
- ST2 empty 0.0
- ST3 empty 0.0
- ST4 empty 0.0
- ST5 empty 0.0
- ST6 empty 0.0
- ST7 empty 0.0

**Registers (Control) window:**

- FST 3100 Cond 0 0 0 1 Err 0 0 0 0 0 0
- FCW 027F Prec NEAR,53 Mask 1 1 1 1
- Last cmd 0023:00A91008 d\_max.00A91008

**Memory dump (Address Hex dump):**

```

Address Hex dump
00A93000 25 66 0A 00 25 66 0A 00 00 00 00 00 00 00 00 00
00A93010 FF FF FF FF FF FF FF FF 00 00 00 00 00 00 00 00
00A93020 FE FF FF FF 01 00 00 00 3E 39 37 B8 C4 C6 C8 47
00A93030 01 00 00 00 48 28 58 00 68 4E 58 00 00 00 00 00
00A93040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A93050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A93060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A93070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A93080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A93090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A930A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A930B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A930C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A930D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A930E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A930F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A93100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A93110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  
```

**Registers (Control) window (bottom):**

- 0021FC60 00A9100C <M RETURN from d\_max
- 0021FC64 33333333 3333
- 0021FC68 3FF33333 33e9
- 0021FC6C 33333333 3333
- 0021FC70 400B3333 33e0
- 0021FC74 00000001 0
- 0021FC78 00A911ED 0
- 0021FC7C 00000001 0
- 0021FC80 00584E68 hNX ASCII "bNX"
- 0021FC84 00582848 hX
- 0021FC88 0816C583 T+7
- 0021FC8C 00000000
- 0021FC90 00000000
- 0021FC94 7EFD0000 pH"
- 0021FC98 00000000
- 0021FC9C 00000000
- 0021FCA0 0021FC88 hX
- 0021FCA4 6F0794F8 07.0
- 0021FCA8 0021FCF4 hX Pointer to next SE handler
- 0021FCAC 00A91639 9.0

Рис. 1.79: OllyDbg: FNSTSW исполнялась

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



TEST работала:

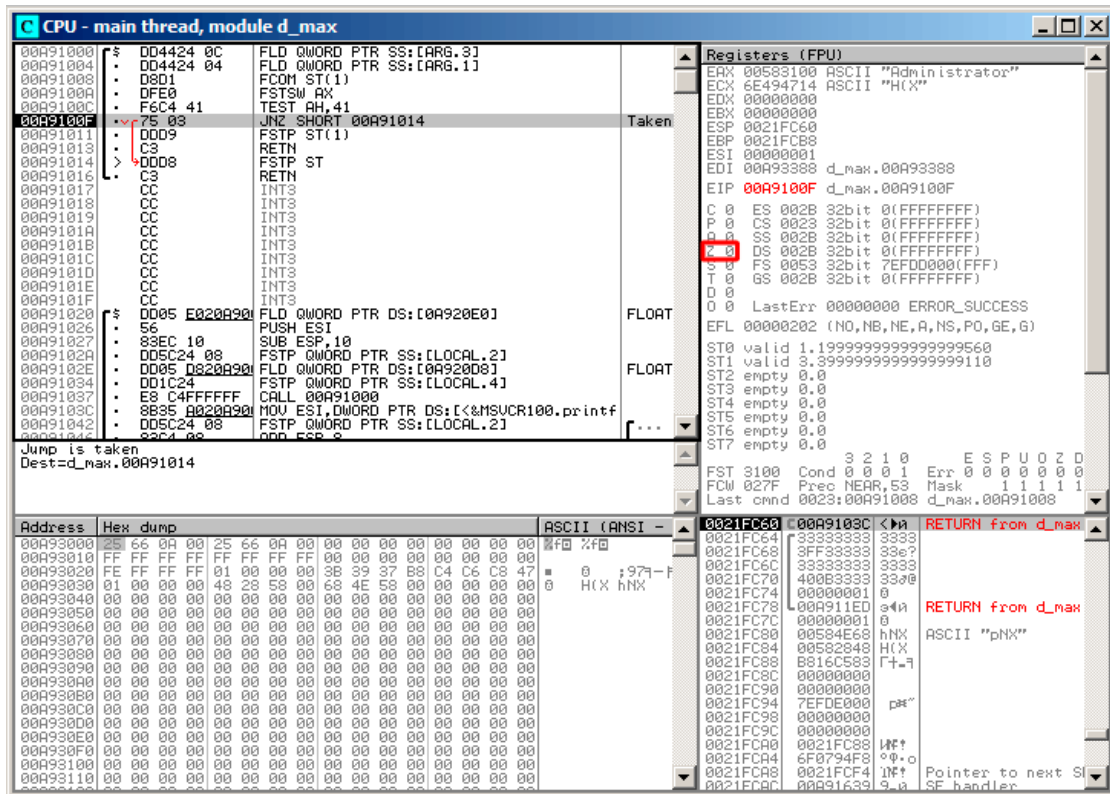


Рис. 1.80: OllyDbg: TEST исполнялась

ZF=0, переход сейчас произойдет.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



## Второй пример с OllyDbg: a=5,6 и b=-4

Обе FLD отработали:

The screenshot shows the OllyDbg interface with the following details:

- Registers (FPU):** ST0 valid 5.5999999999999996440, ST1 valid -4.00000000000000000000. Other registers are empty or contain specific values like EAX=0, ECX=6E445617, etc.
- Disassembly:**
  - Address 00A91020: DD05 F020A90 FLD QWORD PTR DS:[0A920E0]
  - Address 00A91022: DD05 0820A90 FLD QWORD PTR DS:[0A920D8]
- Status Bar:** DD05 F020A90 FLD QWORD PTR DS:[0A920E0]

Рис. 1.82: OllyDbg: обе FLD исполнились

Сейчас будет исполняться FCOM.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

FCOM работала:

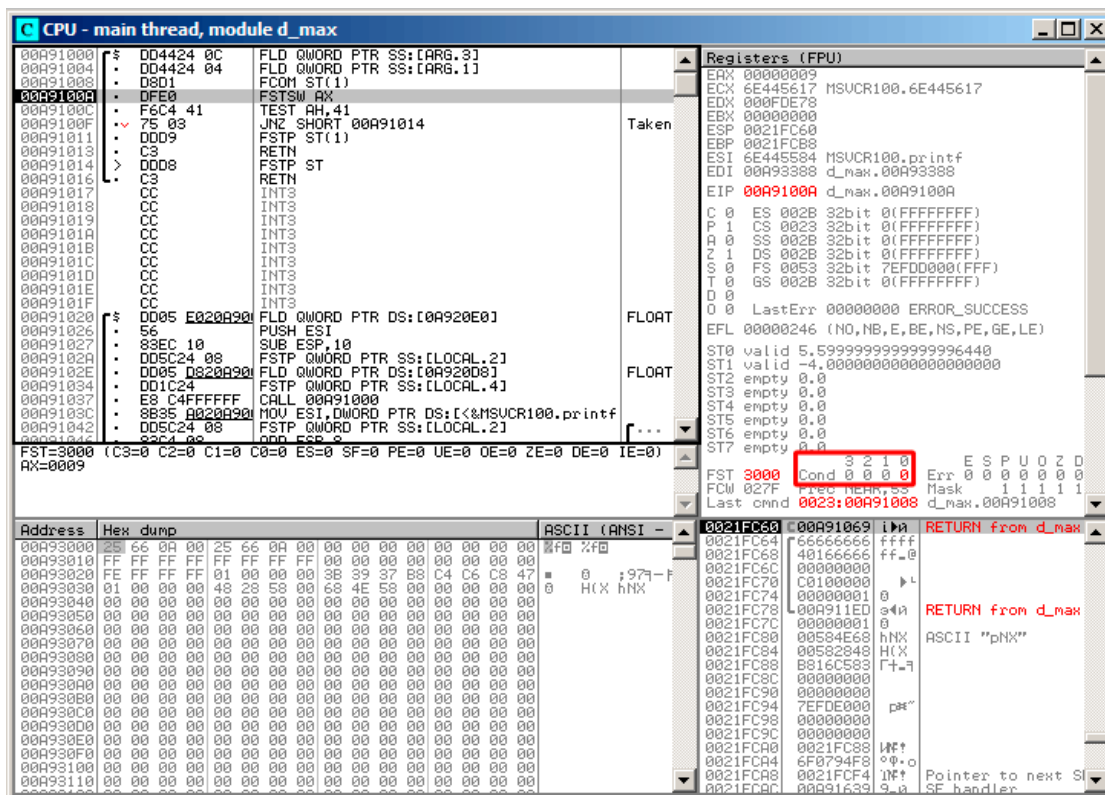


Рис. 1.83: OllyDbg: FCOM исполнялась

Все condition-флаги сброшены.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

FNSTSW сработала, AX=0x3000:

**CPU - main thread, module d\_max**

Address	Hex dump	Instruction	Comment
00A91000	DD4424 0C	FLD QWORD PTR SS:[ARG.3]	
00A91004	DD4424 04	FLD QWORD PTR SS:[ARG.1]	
00A91008	D8D1	FCOM ST(1)	
00A9100C	DFE0	FSTSW AX	
00A91000	F6C4 41	TEST AH, 41	
00A9100F	75 03	JNZ SHORT 00A91014	Taken
00A91011	DD09	FSTP ST(1)	
00A91013	C3	RETN	
00A91014	DD08	FSTP ST	
00A91016	C3	RETN	
00A91017	CC	INT3	
00A91018	CC	INT3	
00A91019	CC	INT3	
00A9101A	CC	INT3	
00A9101B	CC	INT3	
00A9101C	CC	INT3	
00A9101D	CC	INT3	
00A9101E	CC	INT3	
00A9101F	CC	INT3	
00A91020	DD05 F020A900	FLD QWORD PTR DS:[0A920E0]	FLOAT
00A91026	56	PUSH ESI	
00A91027	83EC 10	SUB ESP, 10	
00A9102A	DD5C24 08	FSTP QWORD PTR SS:[LOCAL.2]	FLOAT
00A9102E	DD05 0820A900	FLD QWORD PTR DS:[0A920D8]	FLOAT
00A91034	DD1C24	FSTP QWORD PTR SS:[LOCAL.4]	
00A91037	E8 C4FFFFFF	CALL 00A91000	
00A9103C	8B35 0020A900	MOV ESI, DWORD PTR DS:[<&MSUCR100.printf]	
00A91042	DD5C24 08	FSTP QWORD PTR SS:[LOCAL.2]	
00A91046	83C4 02	ADD ESP, 2	

**Registers (FPU)**

Register	Value
EAX	00003000
MSUCR100	6E445617
EDX	000FDE75
EBX	00000000
ESP	0021FC60
EBP	0021FCB8
ESI	6E445584 MSUCR100.printf
EDI	00A93388 d_max.00A93388
EIP	00A9100C d_max.00A9100C

**Memory Dump**

Address	Hex	ASCII (ANSI)
00A93000	25 66 0A 00 25 66 0A 00 00 00 00 00 00 00 00 00	%f0 %f0
00A93010	FF FF FF FF FF FF FF FF 00 00 00 00 00 00 00 00	
00A93020	FE FF FF FF 01 00 00 00 3B 39 37 B8 C4 C6 C8 47	0 0 ;97-F
00A93030	01 00 00 00 48 28 58 00 68 4E 58 00 00 00 00 00	0 H(X) hNX
00A93040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A93050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A93060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A93070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A93080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A93090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A930A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A930B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A930C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A930D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A930E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A930F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A93100	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A93110	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

**Registers (FPU) - Detail**

Register	Value
ST0	valid 5.5999999999999996440
ST1	valid -4.00000000000000000000
ST2	empty 0.0
ST3	empty 0.0
ST4	empty 0.0
ST5	empty 0.0
ST6	empty 0.0
ST7	empty 0.0

**Memory Dump - Detail**

Address	Hex	ASCII
0021FC60	00A91069	RETURN from d_max
0021FC64	66666666	ffff
0021FC68	40166666	ff_0
0021FC6C	00000000	
0021FC70	C0100000	0
0021FC74	00000001	0
0021FC78	00A911ED	0
0021FC7C	00000001	0
0021FC80	00584E68	hNX
0021FC84	00582848	H(X)
0021FC88	0816C583	Γ=7
0021FC8C	00000000	
0021FC90	00000000	
0021FC94	7EFDE000	p#"
0021FC98	00000000	
0021FC9C	00000000	
0021FCA0	0021FC88	↑↑↑
0021FCA4	6F0794F8	↑↑_o
0021FCA8	0021FCF4	↑↑↑
0021FCAC	00A91639	9_u

Рис. 1.84: OllyDbg: FNSTSW исполнялась

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

TEST работала:

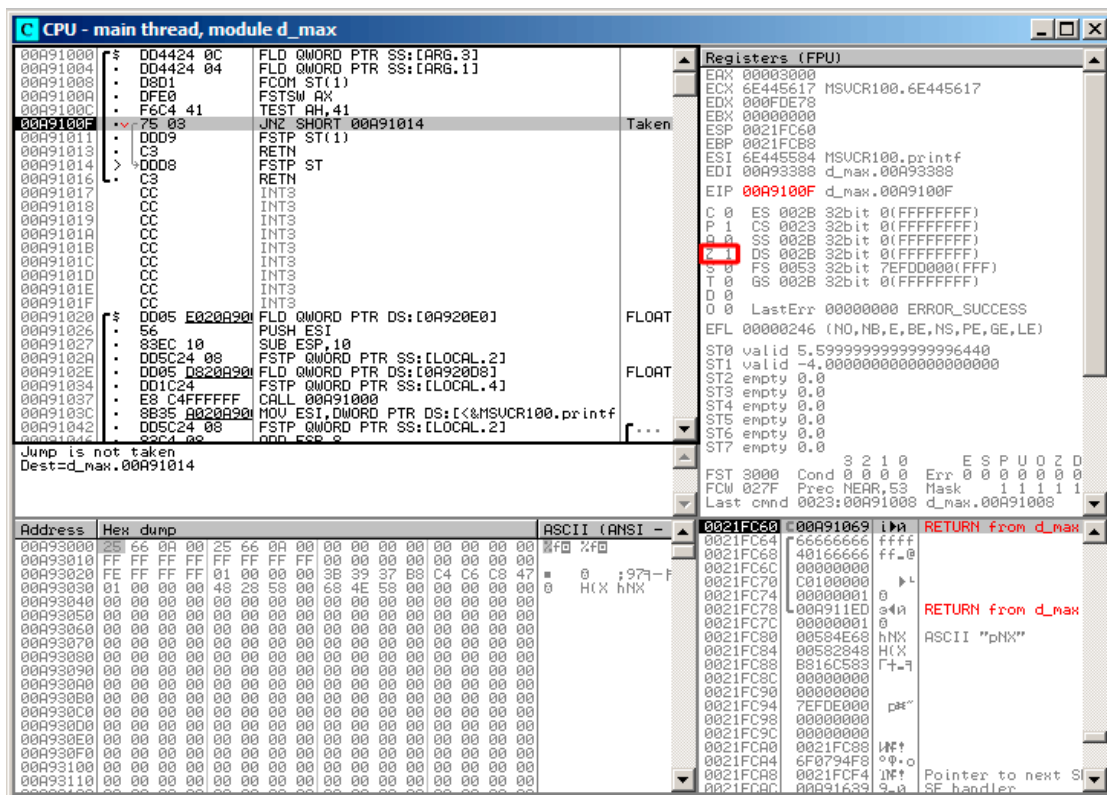


Рис. 1.85: OllyDbg: TEST исполнилась

ZF=1, переход сейчас не произойдет.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



FSTP ST(1) сработала: на вершине FPU-стека осталось значение 5,6.

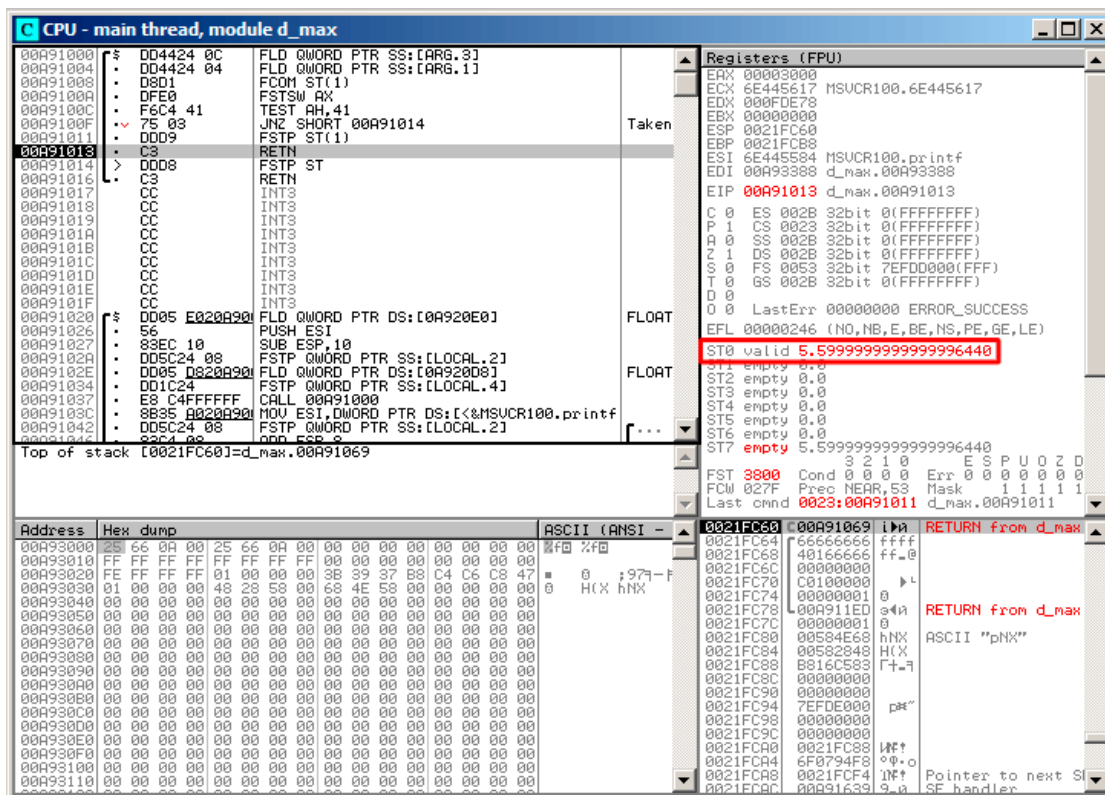


Рис. 1.86: OllyDbg: FSTP исполнялась

Видно, что инструкция FSTP ST(1) работает так: оставляет значение на вершине стека, но обнуляет регистр ST(1).

## GCC 4.4.1

### Листинг 1.216: GCC 4.4.1

```
d_max proc near
b          = qword ptr -10h
a          = qword ptr -8
a_first_half = dword ptr 8
a_second_half = dword ptr 0Ch
b_first_half = dword ptr 10h
b_second_half = dword ptr 14h

push     ebp
mov      ebp, esp
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

sub    esp, 10h
; переложим a и b в локальный стек:

mov    eax, [ebp+a_first_half]
mov    dword ptr [ebp+a], eax
mov    eax, [ebp+a_second_half]
mov    dword ptr [ebp+a+4], eax
mov    eax, [ebp+b_first_half]
mov    dword ptr [ebp+b], eax
mov    eax, [ebp+b_second_half]
mov    dword ptr [ebp+b+4], eax

; загружаем a и b в стек FPU:

fld    [ebp+a]
fld    [ebp+b]

; текущее состояние стека: ST(0) - b; ST(1) - a

fxch  st(1) ; эта инструкция меняет ST(1) и ST(0) местами

; текущее состояние стека: ST(0) - a; ST(1) - b

fucomp ; сравнить a и b и выдернуть из стека два значения, т.е. a и b
fnstsw ax ; записать статус FPU в AX
sahf    ; загрузить состояние флагов SF, ZF, AF, PF, и CF из AH
setnbe al ; записать 1 в AL, если CF=0 и ZF=0
test   al, al ; AL==0 ?
jz     short loc_8048453 ; да
fld    [ebp+a]
jmp    short locret_8048456

loc_8048453:
fld    [ebp+b]

locret_8048456:
leave
retn
d_max endp

```

FUCOMPP — это почти то же что и FCOM, только выкидывает из стека оба значения после сравнения, а также несколько иначе реагирует на «не-числа».

Немного о *не-числах*.

FPU умеет работать со специальными переменными, которые числами не являются и называются «не числа» или **NaN**. Это бесконечность, результат деления на ноль, и так далее. Нечисла бывают «тихие» и «сигнализирующие». С первыми можно продолжать работать и далее, а вот если вы попытаетесь совершить какую-то операцию с сигнализирующим нечислом, то сработает исключение.

Так вот, FCOM вызовет исключение если любой из операндов какое-либо нечисло. FUCOM же вызовет исключение только если один из операндов именно «сиг-

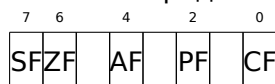
---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

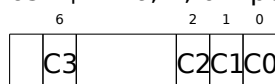


нализирующее нечисло».

Далее мы видим SAHF (*Store AH into Flags*) — это довольно редкая инструкция в коде, не использующим FPU. 8 бит из AH переключаются в младшие 8 бит регистра статуса процессора в таком порядке:



Вспомним, что FNSTSW перегружает интересующие нас биты C3/C2/C0 в AH, и соответственно они будут в позициях 6, 2, 0 в регистре AH:



Иными словами, пара инструкций `fnstsw ax / sahf` переключает биты C3/C2/C0 в флаги ZF, PF, CF.

Теперь снова вспомним, какие значения бит C3/C2/C0 будут при каких результатах сравнения:

- Если  $a$  больше  $b$  в нашем случае, то биты C3/C2/C0 должны быть выставлены так: 0, 0, 0.
- Если  $a$  меньше  $b$ , то биты будут выставлены так: 0, 0, 1.
- Если  $a = b$ , то так: 1, 0, 0.

Иными словами, после трех инструкций `FUCOMPP/FNSTSW/SAHF` возможны такие состояния флагов:

- Если  $a > b$  в нашем случае, то флаги будут выставлены так: ZF=0, PF=0, CF=0.
- Если  $a < b$ , то флаги будут выставлены так: ZF=0, PF=0, CF=1.
- Если  $a = b$ , то так: ZF=1, PF=0, CF=0.

Инструкция `SETNBE` выставит в AL единицу или ноль в зависимости от флагов и условий. Это почти аналог `JNBE`, за тем лишь исключением, что `SETcc`<sup>119</sup> выставляет 1 или 0 в AL, а `Jcc` делает переход или нет. `SETNBE` запишет 1 только если CF=0 и ZF=0. Если это не так, то запишет 0 в AL.

CF будет 0 и ZF будет 0 одновременно только в одном случае: если  $a > b$ .

Тогда в AL будет записана 1, последующий условный переход `JZ` выполнен не будет и функция вернет `_a`. В остальных случаях, функция вернет `_b`.

### Оптимизирующий GCC 4.4.1

Листинг 1.217: Оптимизирующий GCC 4.4.1

```
public d_max
proc near
d_max
```

<sup>119</sup>cc это *condition code*

```

arg_0      = qword ptr 8
arg_8      = qword ptr 10h

        push    ebp
        mov     ebp, esp
        fld    [ebp+arg_0] ; _a
        fld    [ebp+arg_8] ; _b

; состояние стека сейчас: ST(0) = _b, ST(1) = _a
        fxch   st(1)

; состояние стека сейчас: ST(0) = _a, ST(1) = _b
        fucom  st(1) ; сравнить _a и _b
        fnstsw ax
        sahf
        ja     short loc_8048448

; записать ST(0) в ST(0) (холостая операция),
; выкинуть значение лежащее на вершине стека,
; оставить _b на вершине стека
        fstp   st
        jmp    short loc_804844A

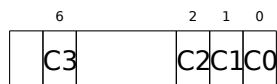
loc_8048448:
; записать _a в ST(1), выкинуть значение лежащее на вершине стека, оставить
; _a на вершине стека
        fstp   st(1)

loc_804844A:
        pop     ebp
        retn
d_max    endp

```

Почти всё что здесь есть, уже описано мною, кроме одного: использование JA после SAHF. Действительно, инструкции условных переходов «больше», «меньше» и «равно» для сравнения беззнаковых чисел (а это JA, JAE, JB, JBE, JE/JZ, JNA, JNAE, JNB, JNBE, JNE/JNZ) проверяют только флаги CF и ZF.

Вспомним, как биты C3/C2/C0 располагаются в регистре AH после исполнения FSTSW/FNSTSW:



Вспомним также, как располагаются биты из AH во флагах CPU после исполнения SAHF:



Биты C3 и C0 после сравнения переключаются в флаги ZF и CF так, что переносные инструкции переходов могут работать. работает, если CF и ZF

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

обнулены.

Таким образом, перечисленные инструкции условного перехода можно использовать после инструкций FNSTSW/SAHF.

Может быть, биты статуса FPU C3/C2/C0 преднамеренно были размещены таким образом, чтобы переноситься на базовые флаги процессора без перестановок?

### GCC 4.8.1 с оптимизацией -O3

В линейке процессоров P6 от Intel появились новые FPU-инструкции<sup>120</sup>. Это FUCOMI (сравнить операнды и выставить флаги основного CPU) и FCMOVcc (работает как CMOVcc, но на регистрах FPU). Очевидно, разработчики GCC решили отказаться от поддержки процессоров до линейки P6 (ранние Pentium, 80486, и т. д.).

И кстати, FPU уже давно не отдельная часть процессора в линейке P6, так что флаги основного CPU можно модифицировать из FPU.

Вот что имеем:

Листинг 1.218: Оптимизирующий GCC 4.8.1

```
fld    QWORD PTR [esp+4]      ; загрузить "a"
fld    QWORD PTR [esp+12]    ; загрузить "b"
; ST0=b, ST1=a
fxch   st(1)
; ST0=a, ST1=b
; сравнить "a" и "b"
fucomi st, st(1)
; скопировать ST1 (там "b") в ST0 если a<=b
; в противном случае, оставить "a" в ST0
fcmovbe st, st(1)
; выбросить значение из ST1
fstp   st(1)
ret
```

Не совсем понимаю, зачем здесь FXCH (поменять местами операнды).

От нее легко избавиться поменяв местами инструкции FLD либо заменив FCMOVBE (*below or equal* — меньше или равно) на FCMOVA (*above* — больше).

Должно быть, неаккуратность компилятора.

Так что FUCOMI сравнивает ST(0) (*a*) и ST(1) (*b*) и затем устанавливает флаги основного CPU. FCMOVBE проверяет флаги и копирует ST(1) (в тот момент там находится *b*) в ST(0) (там *a*) если  $ST0(a) \leq ST1(b)$ . В противном случае ( $a > b$ ), она оставляет *a* в ST(0).

Последняя FSTP оставляет содержимое ST(0) на вершине стека, выбрасывая содержимое ST(1).

Попробуем оттрассировать функцию в GDB:

<sup>120</sup>Начиная с Pentium Pro, Pentium-II, и т. д.

## Листинг 1.219: Оптимизирующий GCC 4.8.1 and GDB

```

1 dennis@ubuntuvm:~/polygon$ gcc -O3 d_max.c -o d_max -fno-inline
2 dennis@ubuntuvm:~/polygon$ gdb d_max
3 GNU gdb (GDB) 7.6.1-ubuntu
4 ...
5 Reading symbols from /home/dennis/polygon/d_max...(no debugging symbols ↵
   ↵ found)...done.
6 (gdb) b d_max
7 Breakpoint 1 at 0x80484a0
8 (gdb) run
9 Starting program: /home/dennis/polygon/d_max
10
11 Breakpoint 1, 0x080484a0 in d_max ()
12 (gdb) ni
13 0x080484a4 in d_max ()
14 (gdb) disas $eip
15 Dump of assembler code for function d_max:
16   0x080484a0 <+0>:   fldl   0x4(%esp)
17 => 0x080484a4 <+4>:   fldl   0xc(%esp)
18   0x080484a8 <+8>:   fxch  %st(1)
19   0x080484aa <+10>:  fucomi %st(1),%st
20   0x080484ac <+12>:  fcmovbe %st(1),%st
21   0x080484ae <+14>:  fstp  %st(1)
22   0x080484b0 <+16>:  ret
23 End of assembler dump.
24 (gdb) ni
25 0x080484a8 in d_max ()
26 (gdb) info float
27   R7: Valid   0x3fff999999999999800 +1.19999999999999956
28 =>R6: Valid   0x4000d999999999999800 +3.39999999999999911
29   R5: Empty   0x00000000000000000000
30   R4: Empty   0x00000000000000000000
31   R3: Empty   0x00000000000000000000
32   R2: Empty   0x00000000000000000000
33   R1: Empty   0x00000000000000000000
34   R0: Empty   0x00000000000000000000
35
36 Status Word:      0x3000
37                   TOP: 6
38 Control Word:     0x037f   IM DM ZM OM UM PM
39                   PC: Extended Precision (64-bits)
40                   RC: Round to nearest
41 Tag Word:         0x0fff
42 Instruction Pointer: 0x73:0x080484a4
43 Operand Pointer:   0x7b:0xbffff118
44 Opcode:           0x0000
45 (gdb) ni
46 0x080484aa in d_max ()
47 (gdb) info float
48   R7: Valid   0x4000d999999999999800 +3.39999999999999911
49 =>R6: Valid   0x3fff9999999999999800 +1.19999999999999956
50   R5: Empty   0x00000000000000000000
51   R4: Empty   0x00000000000000000000

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

52  R3: Empty  0x00000000000000000000
53  R2: Empty  0x00000000000000000000
54  R1: Empty  0x00000000000000000000
55  R0: Empty  0x00000000000000000000
56
57  Status Word:      0x3000
58                    TOP: 6
59  Control Word:    0x037f  IM DM ZM OM UM PM
60                    PC: Extended Precision (64-bits)
61                    RC: Round to nearest
62  Tag Word:        0x0fff
63  Instruction Pointer: 0x73:0x080484a8
64  Operand Pointer:  0x7b:0xbffff118
65  Opcode:          0x0000
66  (gdb) disas $eip
67  Dump of assembler code for function d_max:
68      0x080484a0 <+0>:   fldl   0x4(%esp)
69      0x080484a4 <+4>:   fldl   0xc(%esp)
70      0x080484a8 <+8>:   fxch  %st(1)
71  => 0x080484aa <+10>:  fucomi %st(1),%st
72      0x080484ac <+12>:  fcmovbe %st(1),%st
73      0x080484ae <+14>:  fstp  %st(1)
74      0x080484b0 <+16>:  ret
75  End of assembler dump.
76  (gdb) ni
77  0x080484ac in d_max ()
78  (gdb) info registers
79  eax          0x1          1
80  ecx          0xbffff1c4      -1073745468
81  edx          0x8048340       134513472
82  ebx          0xb7bf000       -1208225792
83  esp          0xbffff10c      0xbffff10c
84  ebp          0xbffff128      0xbffff128
85  esi          0x0            0
86  edi          0x0            0
87  eip          0x80484ac       0x80484ac <d_max+12>
88  eflags      0x203          [ CF IF ]
89  cs          0x73           115
90  ss          0x7b           123
91  ds          0x7b           123
92  es          0x7b           123
93  fs          0x0            0
94  gs          0x33           51
95  (gdb) ni
96  0x080484ae in d_max ()
97  (gdb) info float
98  R7: Valid  0x4000d99999999999800 +3.39999999999999911
99  =>R6: Valid 0x4000d99999999999800 +3.39999999999999911
100 R5: Empty  0x00000000000000000000
101 R4: Empty  0x00000000000000000000
102 R3: Empty  0x00000000000000000000
103 R2: Empty  0x00000000000000000000
104 R1: Empty  0x00000000000000000000

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

105 R0: Empty 0x00000000000000000000
106
107 Status Word: 0x3000
108 TOP: 6
109 Control Word: 0x037f IM DM ZM OM UM PM
110 PC: Extended Precision (64-bits)
111 RC: Round to nearest
112 Tag Word: 0x0fff
113 Instruction Pointer: 0x73:0x080484ac
114 Operand Pointer: 0x7b:0xbffff118
115 Opcode: 0x0000
116 (gdb) disas $eip
117 Dump of assembler code for function d_max:
118 0x080484a0 <+0>: fldl 0x4(%esp)
119 0x080484a4 <+4>: fldl 0xc(%esp)
120 0x080484a8 <+8>: fxch %st(1)
121 0x080484aa <+10>: fucomi %st(1),%st
122 0x080484ac <+12>: fcmovbe %st(1),%st
123 => 0x080484ae <+14>: fstp %st(1)
124 0x080484b0 <+16>: ret
125 End of assembler dump.
126 (gdb) ni
127 0x080484b0 in d_max ()
128 (gdb) info float
129 =>R7: Valid 0x4000d9999999999999800 +3.39999999999999911
130 R6: Empty 0x4000d9999999999999800
131 R5: Empty 0x000000000000000000000000
132 R4: Empty 0x000000000000000000000000
133 R3: Empty 0x000000000000000000000000
134 R2: Empty 0x000000000000000000000000
135 R1: Empty 0x000000000000000000000000
136 R0: Empty 0x000000000000000000000000
137
138 Status Word: 0x3800
139 TOP: 7
140 Control Word: 0x037f IM DM ZM OM UM PM
141 PC: Extended Precision (64-bits)
142 RC: Round to nearest
143 Tag Word: 0x3fff
144 Instruction Pointer: 0x73:0x080484ae
145 Operand Pointer: 0x7b:0xbffff118
146 Opcode: 0x0000
147 (gdb) quit
148 A debugging session is active.
149
150 Inferior 1 [process 30194] will be killed.
151
152 Quit anyway? (y or n) y
153 dennis@ubuntuvms:~/polygon$

```

Используя «ni», дадим первым двум инструкциям FLD исполниться.

Посмотрим регистры FPU (строка 33).

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Как уже было указано ранее, регистры FPU это скорее кольцевой буфер, нежели стек (?? (стр. ??)). И GDB показывает не регистры STx, а внутренние регистры FPU (Rx). Стрелка (на строке 35) указывает на текущую вершину стека.

Вы можете также увидеть содержимое регистра TOP в «Status Word» (строка 36-37). Там сейчас 6, так что вершина стека сейчас указывает на внутренний регистр 6.

Значения *a* и *b* меняются местами после исполнения FXCH (строка 54).

FUCOMI исполнилась (строка 83). Посмотрим флаги: CF выставлен (строка 95).

FCMOVBE действительно скопировал значение *b* (см. строку 104).

FSTP оставляет одно значение на вершине стека (строка 139). Значение TOP теперь 7, так что вершина FPU-стека указывает на внутренний регистр 7.

## ARM

### Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

Листинг 1.220: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

VMOV	D16, R2, R3 ; b
VMOV	D17, R0, R1 ; a
VCMPE.F64	D17, D16
VMRS	APSR_nzcv, FPSCR
VMOVGT.F64	D16, D17 ; скопировать "a" в D16
VMOV	R0, R1, D16
BX	LR

Очень простой случай. Входные величины помещаются в D17 и D16 и сравниваются при помощи инструкции VCMPE. Как и в сопроцессорах x86, сопроцессор в ARM имеет свой собственный регистр статуса и флагов (**FPSCR!**<sup>121</sup>), потому что есть необходимость хранить специфичные для его работы флаги.

И так же, как и в x86, в ARM нет инструкций условного перехода, проверяющих биты в регистре статуса сопроцессора. Поэтому имеется инструкция VMRS, копирующая 4 бита (N, Z, C, V) из статуса сопроцессора в биты *общего* статуса (регистр **APSR!**<sup>122</sup>).

VMOVGT это аналог MOVGT, инструкция для D-регистров, срабатывающая, если при сравнении один операнд был больше чем второй (*GT* — *Greater Than*).

Если она сработает, в D16 запишется значение *a*, лежащее в тот момент в D17. В обратном случае в D16 остается значение *b*.

Предпоследняя инструкция VMOV готовит то, что было в D16, для возврата через пару регистров R0 и R1.

<sup>121</sup>**FPSCR!**

<sup>122</sup>**APSR!**

**Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)**

Листинг 1.221: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

VMOV	D16, R2, R3 ; b
VMOV	D17, R0, R1 ; a
VCMPE.F64	D17, D16
VMRS	APSR_nzcv, FPSCR
IT GT	
VMOVGT.F64	D16, D17
VMOV	R0, R1, D16
BX	LR

Почти то же самое, что и в предыдущем примере, за парой отличий. Как мы уже знаем, многие инструкции в режиме ARM можно дополнять условием. Но в режиме Thumb такого нет. В 16-битных инструкций просто нет места для лишних 4 битов, при помощи которых можно было бы закодировать условие выполнения.

Поэтому в Thumb-2 добавили возможность дополнять Thumb-инструкции условиями. В листинге, сгенерированном при помощи **IDA!**, мы видим инструкцию VMOVGT, такую же как и в предыдущем примере.

В реальности там закодирована обычная инструкция VMOV, просто **IDA!** добавила суффикс -GT к ней, потому что перед этой инструкцией стоит IT GT.

Инструкция IT определяет так называемый *if-then block*. После этой инструкции можно указывать до четырех инструкций, к каждой из которых будет добавлен суффикс условия.

В нашем примере IT GT означает, что следующая за ней инструкция будет исполнена, если условие GT (*Greater Than*) справедливо.

Теперь более сложный пример. Кстати, из Angry Birds (для iOS):

Листинг 1.222: Angry Birds Classic

...	
ITE NE	
VMOVNE	R2, R3, D16
VMOVEQ	R2, R3, D17
BLX	_objc_msgSend ; без суффикса
...	

ITE означает *if-then-else* и кодирует суффиксы для двух следующих за ней инструкций.

Первая из них исполнится, если условие, закодированное в ITE (*NE, not equal*) будет в тот момент справедливо, а вторая — если это условие не сработает. (Обратное условие от NE это EQ (*equal*)).

Инструкция следующая за второй VMOV (или VMOEQ) нормальная, без суффикса (BLX).

Ещё чуть сложнее, и снова этот фрагмент из Angry Birds:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



## Листинг 1.223: Angry Birds Classic

```

...
ITTTT EQ
MOVEQ      R0, R4
ADDEQ      SP, SP, #0x20
POPEQ.W    {R8,R10}
POPEQ      {R4-R7,PC}
BLX        ___stack_chk_fail ; без суффикса
...

```

Четыре символа «Т» в инструкции означают, что четыре последующие инструкции будут выполнены если условие соблюдается. Поэтому **IDA!** добавила ко всем четырем инструкциям суффикс -EQ. А если бы здесь было, например, ITEEE EQ (*if-then-else-else-else*), тогда суффиксы для следующих четырех инструкций были бы расставлены так:

```

-EQ
-NE
-NE
-NE

```

Ещё фрагмент из Angry Birds:

## Листинг 1.224: Angry Birds Classic

```

...
CMP.W      R0, #0xFFFFFFFF
ITTE LE
SUBLE.W    R10, R0, #1
NEGLE      R0, R0
MOVGT      R10, R0
MOVS       R6, #0 ; без суффикса
CBZ        R0, loc_1E7E32 ; без суффикса
...

```

ITTE (*if-then-then-else*) означает, что первая и вторая инструкции выполняются, если условие LE (*Less or Equal*) справедливо, а третья — если справедливо обратное условие (GT — *Greater Than*).

Компиляторы способны генерировать далеко не все варианты.

Например, в вышеупомянутой игре Angry Birds (версия *classic* для iOS)

встречаются только такие варианты инструкции IT: IT, ITE, ITT, ITTE, ITTT, ITTTT. Как это узнать? В **IDA!** можно сгенерировать листинг (что и было сделано), только в опциях был установлен показ 4 байтов для каждого опкода.

Затем, зная что старшая часть 16-битного опкода (IT это 0xBF), сделаем при помощи `grep` это:

```
cat AngryBirdsClassic.lst | grep " BF" | grep "IT" > results.lst
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Кстати, если писать на ассемблере для режима Thumb-2 вручную, и дополнять инструкции суффиксами условия, то ассемблер автоматически будет добавлять инструкцию IT с соответствующими флагами там, где надо.

### Неоптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

Листинг 1.225: Неоптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

```

b          = -0x20
a          = -0x18
val_to_return = -0x10
saved_R7   = -4

          STR      R7, [SP,#saved_R7]!
          MOV      R7, SP
          SUB      SP, SP, #0x1C
          BIC      SP, SP, #7
          VMOV     D16, R2, R3
          VMOV     D17, R0, R1
          VSTR     D17, [SP,#0x20+a]
          VSTR     D16, [SP,#0x20+b]
          VLDR     D16, [SP,#0x20+a]
          VLDR     D17, [SP,#0x20+b]
          VCMPE.F64 D16, D17
          VMRS     APSR_nzcv, FPSCR
          BLE     loc_2E08
          VLDR     D16, [SP,#0x20+a]
          VSTR     D16, [SP,#0x20+val_to_return]
          B       loc_2E10

loc_2E08   VLDR     D16, [SP,#0x20+b]
          VSTR     D16, [SP,#0x20+val_to_return]

loc_2E10   VLDR     D16, [SP,#0x20+val_to_return]
          VMOV     R0, R1, D16
          MOV      SP, R7
          LDR      R7, [SP+0x20+b], #4
          BX      LR

```

Почти то же самое, что мы уже видели, но много избыточного кода из-за хранения *a* и *b*, а также выходного значения, в локальном стеке.

### Оптимизирующий Keil 6/2013 (Режим Thumb)

Листинг 1.226: Оптимизирующий Keil 6/2013 (Режим Thumb)

```

          PUSH     {R3-R7, LR}
          MOVS    R4, R2
          MOVS    R5, R3

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мой адреса](#). Спасибо!

```

        MOVS    R6, R0
        MOVS    R7, R1
        BL      __aeabi_cdrcmple
        BCS     loc_1C0
        MOVS    R0, R6
        MOVS    R1, R7
        POP     {R3-R7, PC}

loc_1C0
        MOVS    R0, R4
        MOVS    R1, R5
        POP     {R3-R7, PC}

```

Keil не генерирует FPU-инструкции, потому что не рассчитывает на то, что они будут поддерживаться, а простым сравнением побитово здесь не обойтись.

Для сравнения вызывается библиотечная функция `__aeabi_cdrcmple`.

N.B. Результат сравнения эта функция оставляет в флагах, чтобы следующая за вызовом инструкция *BCS* (*Carry set — Greater than or equal*) могла работать без дополнительного кода.

## ARM64

### Оптимизирующий GCC (Linaro) 4.9

```

d_max:
; D0 - a, D1 - b
    fcmpe    d0, d1
    fcsel    d0, d0, d1, gt
; теперь результат в D0
    ret

```

В ARM64 **ISA!** теперь есть FPU-инструкции, устанавливающие флаги CPU **APSR!** вместо **FPSCR!** для удобства. **FPU!** больше не отдельное устройство (по крайней мере логически). Это FCMPE. Она сравнивает два значения, переданных в D0 и D1 (а это первый и второй аргументы функции) и выставляет флаги в **APSR!** (N, Z, C, V).

FCSEL (*Floating Conditional Select*) копирует значение D0 или D1 в D0 в зависимости от условия (GT — Greater Than — больше чем), и снова, она использует флаги в регистре **APSR!** вместо **FPSCR!**. Это куда удобнее, если сравнивать с тем набором инструкций, что был в процессорах раньше.

Если условие верно (GT), тогда значение из D0 копируется в D0 (т.е. ничего не происходит). Если условие не верно, то значение D1 копируется в D0.

### Неоптимизирующий GCC (Linaro) 4.9

```

d_max:

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; сохранить входные аргументы в "Register Save Area"
    sub    sp, sp, #16
    str    d0, [sp,8]
    str    d1, [sp]
; перезагрузить значения
    ldr    x1, [sp,8]
    ldr    x0, [sp]
    fmov   d0, x1
    fmov   d1, x0
; D0 - a, D1 - b
    fcmpe  d0, d1
    ble    .L76
; a>b; загрузить D0 (a) в X0
    ldr    x0, [sp,8]
    b      .L74
.L76:
; a<=b; загрузить D1 (b) в X0
    ldr    x0, [sp]
.L74:
; результат в X0
    fmov   d0, x0
; результат в D0
    add    sp, sp, 16
    ret

```

Неоптимизирующий GCC более многословен. В начале функция сохраняет значения входных аргументов в локальном стеке (*Register Save Area*). Затем код перезагружает значения в регистры X0/X1 и наконец копирует их в D0/D1 для сравнения инструкцией FCMPE. Много избыточного кода, но так работают неоптимизирующие компиляторы. FCMPE сравнивает значения и устанавливает флаги в **APSR!**. В этот момент компилятор ещё не думает о более удобной инструкции FCSEL, так что он работает старым методом: использует инструкцию BLE (*Branch if Less than or Equal* (переход если меньше или равно)). В одном случае ( $a > b$ ) значение  $a$  перезагружается в X0. В другом случае ( $a \leq b$ ) значение  $b$  загружается в X0. Наконец, значение из X0 копируется в D0, потому что возвращаемое значение оставляется в этом регистре.

### Упражнение

Для упражнения вы можете попробовать оптимизировать этот фрагмент кода вручную, удалив избыточные инструкции, но не добавляя новых (включая FCSEL).

### Оптимизирующий GCC (Linaro) 4.9: float

Перепишем пример. Теперь здесь *float* вместо *double*.

```

float f_max (float a, float b)
{
    if (a>b)

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        return a;

    return b;
};

```

```

f_max:
; S0 - a, S1 - b
    fcmpe    s0, s1
    fcsel    s0, s0, s1, gt
; теперь результат в S0
    ret

```

Всё то же самое, только используются S-регистры вместо D-. Так что числа типа *float* передаются в 32-битных S-регистрах (а это младшие части 64-битных D-регистров).

## MIPS

В сопроцессоре MIPS есть бит результата, который устанавливается в FPU и проверяется в CPU.

Ранние MIPS имели только один бит (с названием FCC0), а у поздних их 8 (с названием FCC7-FCC0). Этот бит (или биты) находятся в регистре с названием FCCR.

### Листинг 1.227: Оптимизирующий GCC 4.4.5 (IDA)

```

d_max:
; установить бит условия FPU в 1, если $f14<$f12 (b<a):
    c.lt.d   $f14, $f12
    or      $at, $zero ; NOP
; перейти на locret_14 если бит условия выставлен
    bc1t    locret_14
; эта инструкция всегда исполняется (установить значение для возврата в "a"):
    mov.d   $f0, $f12 ; branch delay slot
; эта инструкция исполняется только если переход не произошел
; (т.е., если b>=a)
; установить значение для возврата в "b":
    mov.d   $f0, $f14

locret_14:
    jr      $ra
    or     $at, $zero ; branch delay slot, NOP

```

C.LT.D сравнивает два значения. LT это условие «Less Than» (меньше чем). D означает переменные типа *double*.

В зависимости от результата сравнения, бит FCC0 устанавливается или очищается.

BC1T проверяет бит FCC0 и делает переход, если бит выставлен. T означает, что переход произойдет если бит выставлен («True»). Имеется также инструкция BC1F которая сработает, если бит сброшен («False»).

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

В зависимости от перехода один из аргументов функции помещается в регистр \$F0.

### 1.25.8. Некоторые константы

В Wikipedia легко найти представление некоторых констант в IEEE 754. Любопытно узнать, что 0.0 в IEEE 754 представляется как 32 нулевых бита (для одинарной точности) или 64 нулевых бита (для двойной). Так что, для записи числа 0.0 в переменную в памяти или регистр, можно пользоваться инструкцией MOV, или XOR reg, reg. Это тем может быть удобно, что если в структуре есть много переменных разных типов, то обычной ф-цией memset() можно установить все целочисленные переменные в 0, все булевы переменные в false, все указатели в NULL, и все переменные с плавающей точкой (любой точности) в 0.0.

### 1.25.9. Копирование

По инерции можно подумать, что для загрузки и сохранения (и, следовательно, копирования) чисел в формате IEEE 754 нужно использовать пару инструкций FLD/FST. Тем не менее, этого куда легче достичь используя обычную инструкцию MOV, которая, конечно же, просто копирует значения побитово.

### 1.25.10. Стек, калькуляторы и обратная польская запись

Теперь понятно, почему некоторые старые программируемые калькуляторы используют обратную польскую запись.

Например для сложения 12 и 34 нужно было набрать 12, потом 34, потом нажать знак «плюс».

Это потому что старые калькуляторы просто реализовали стековую машину и это было куда проще, чем обрабатывать сложные выражения со скобками.

Подобный калькулятор все еще присутствует во многих Unix-дистрибутивах: *dc*.

### 1.25.11. 80 бит?

Внутреннее представление чисел с FPU — 80-битное. Странное число, потому как не является числом вида  $2^n$ . Имеется гипотеза, что причина, возможно, историческая — стандартные IBM-овские перфокарты могли кодировать 12 строк по 80 бит. Раньше было также популярно текстовое разрешение 80·25.

В Wikipedia есть еще одно объяснение: [https://en.wikipedia.org/wiki/Extended\\_precision](https://en.wikipedia.org/wiki/Extended_precision).

Если вы знаете более точную причину, просьба сообщить автору: [мои адреса](#).

### 1.25.12. x64

О том, как происходит работа с числами с плавающей запятой в x86-64, читайте здесь: ?? (стр. ??).

### 1.25.13. Упражнения

- <http://challenges.re/60>
- <http://challenges.re/61>

## 1.26. Массивы

Массив это просто набор переменных в памяти, обязательно лежащих рядом и обязательно одного типа<sup>123</sup>.

### 1.26.1. Простой пример

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<20; i++)
        a[i]=i*2;

    for (i=0; i<20; i++)
        printf ("a[%d]=%d\n", i, a[i]);

    return 0;
};
```

### x86

### MSVC

Компилируем:

Листинг 1.228: MSVC 2008

```
_TEXT    SEGMENT
_i$ = -84                ; size = 4
_a$ = -80                ; size = 80
_main    PROC
    push    ebp
    mov     ebp, esp
```

<sup>123</sup> **АКА!** «гомогенный контейнер»

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

sub    esp, 84      ; 00000054H
mov    DWORD PTR _i$[ebp], 0
jmp    SHORT $LN6@main
$LN5@main:
mov    eax, DWORD PTR _i$[ebp]
add    eax, 1
mov    DWORD PTR _i$[ebp], eax
$LN6@main:
cmp    DWORD PTR _i$[ebp], 20 ; 00000014H
jge    SHORT $LN4@main
mov    ecx, DWORD PTR _i$[ebp]
shl   ecx, 1
mov    edx, DWORD PTR _i$[ebp]
mov    DWORD PTR _a$[ebp+edx*4], ecx
jmp    SHORT $LN5@main
$LN4@main:
mov    DWORD PTR _i$[ebp], 0
jmp    SHORT $LN3@main
$LN2@main:
mov    eax, DWORD PTR _i$[ebp]
add    eax, 1
mov    DWORD PTR _i$[ebp], eax
$LN3@main:
cmp    DWORD PTR _i$[ebp], 20 ; 00000014H
jge    SHORT $LN1@main
mov    ecx, DWORD PTR _i$[ebp]
mov    edx, DWORD PTR _a$[ebp+ecx*4]
push  edx
mov    eax, DWORD PTR _i$[ebp]
push  eax
push  OFFSET $SG2463
call  _printf
add   esp, 12      ; 0000000cH
jmp    SHORT $LN2@main
$LN1@main:
xor    eax, eax
mov    esp, ebp
pop    ebp
ret    0
_main  ENDP

```

Ничего особенного, просто два цикла. Один изменяет массив, второй печатает его содержимое. Команда `shl ecx, 1` используется для умножения ECX на 2, об этом: (?? (стр. ??)).

Под массив выделено в стеке 80 байт, это 20 элементов по 4 байта.



Попробуем этот пример в OllyDbg.

Видно, как заполнился массив: каждый элемент это 32-битное слово типа *int*, с шагом 2:

The screenshot shows the OllyDbg interface with the following details:

- Assembly View:** Shows assembly instructions for the main thread. A loop is visible where the instruction `MOV DWORD PTR SS:[LOCAL.21],0` is repeated, filling a stack array. The instruction `JMP SHORT 0040102C` is highlighted.
- Registers (FPU):** Shows the state of CPU registers. EAX is 00000014, ECX is 00000026, EDX is 00000013, EBX is 00000000, ESP is 0018FEF0, EBP is 0018FF44, ESI is 00000001, EDI is 0040338C, and EIP is 0040102C.
- Stack View:** Shows a memory dump of the stack. The address range is from 0018FEF4 to 0018FF44. The hex dump shows a repeating pattern of `00 00 00 02` every 4 bytes, indicating the value 20 is being stored in the array.
- Registers (General Purpose):** Shows the state of general-purpose registers. EAX, ECX, EDI, and ESI are empty (0). EBP, ESP, and EIP are filled with their respective values.
- Registers (Control):** Shows the state of control registers. CS is 00000000, DS is 00000000, SS is 00000000, and FS is 00000000.
- Registers (Status):** Shows the state of status registers. EFL is 00000246, and other status registers are empty.

Рис. 1.87: OllyDbg: после заполнения массива

А так как этот массив находится в стеке, то мы видим все его 20 элементов внутри стека.

## GCC

Рассмотрим результат работы GCC 4.4.1:

### Листинг 1.229: GCC 4.4.1

```
main public main
proc near ; DATA XREF: _start+17
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

var_70      = dword ptr -70h
var_6C      = dword ptr -6Ch
var_68      = dword ptr -68h
i_2        = dword ptr -54h
i           = dword ptr -4

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFF0h
        sub     esp, 70h
        mov     [esp+70h+i], 0           ; i=0
        jmp     short loc_804840A

loc_80483F7:
        mov     eax, [esp+70h+i]
        mov     edx, [esp+70h+i]
        add     edx, edx                 ; edx=i*2
        mov     [esp+eax*4+70h+i_2], edx
        add     [esp+70h+i], 1         ; i++

loc_804840A:
        cmp     [esp+70h+i], 13h
        jle     short loc_80483F7
        mov     [esp+70h+i], 0
        jmp     short loc_8048441

loc_804841B:
        mov     eax, [esp+70h+i]
        mov     edx, [esp+eax*4+70h+i_2]
        mov     eax, offset aADD ; "a[%d]=%d\n"
        mov     [esp+70h+var_68], edx
        mov     edx, [esp+70h+i]
        mov     [esp+70h+var_6C], edx
        mov     [esp+70h+var_70], eax
        call    _printf
        add     [esp+70h+i], 1

loc_8048441:
        cmp     [esp+70h+i], 13h
        jle     short loc_804841B
        mov     eax, 0
        leave
        retn

main      endp

```

Переменная *a* в нашем примере имеет тип *int\** (указатель на *int*). Вы можете попробовать передать в другую функцию указатель на массив, но точнее было бы сказать, что передается указатель на первый элемент массива (а адреса остальных элементов массива можно вычислить очевидным образом).

Если индексировать этот указатель как *a[idx]*, *idx* просто прибавляется к указателю и возвращается элемент, расположенный там, куда ссылается вычисленный указатель.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Вот любопытный пример. Строка символов вроде *string* это массив из символов. Она имеет тип *const char[]*. К этому указателю также можно применять индекс.

Поэтому можно написать даже так: «string»[i] — это совершенно легальное выражение в Си/Си++!

## ARM

### Неоптимизирующий Keil 6/2013 (Режим ARM)

```

EXPORT _main
_main
    STMFD    SP!, {R4,LR}
; выделить место для 20-и переменных типа int:
    SUB     SP, SP, #0x50
; первый цикл:
    MOV     R4, #0           ; i
    B      loc_4A0
loc_494
    MOV     R0, R4,LSL#1     ; R0=R4*2
; сохранить R0 в SP+R4<<2 (то же что и SP+R4*4):
    STR     R0, [SP,R4,LSL#2]
    ADD     R4, R4, #1       ; i=i+1
loc_4A0
    CMP     R4, #20          ; i<20?
    BLT    loc_494          ; да, запустить тело цикла снова
; второй цикл:
    MOV     R4, #0           ; i
    B      loc_4C4
loc_4B0
    LDR     R2, [SP,R4,LSL#2]; (второй аргумент printf)
    R2=*(SP+R4<<4) (то же что и *(SP+R4*4))
    MOV     R1, R4           ; (первый аргумент printf) R1=i
    ADR     R0, aADD         ; "a[%d]=%d\n"
    BL     __2printf
    ADD     R4, R4, #1       ; i=i+1
loc_4C4
    CMP     R4, #20          ; i<20?
    BLT    loc_4B0          ; да, запустить тело цикла снова
    MOV     R0, #0           ; значение для возврата
; освободить блок в стеке, выделенное для 20 переменных:
    ADD     SP, SP, #0x50
    LDMFD   SP!, {R4,PC}

```

Тип *int* требует 32 бита для хранения (или 4 байта),

так что для хранения 20 переменных типа *int*, нужно 80 (0x50) байт.

Поэтому инструкция `SUB SP, SP, #0x50` в прологе функции выделяет в локальном стеке под массив именно столько места.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

И в первом и во втором цикле итератор цикла  $i$  будет постоянно находиться в регистре R4.

Число, которое нужно записать в массив, вычисляется так:  $i * 2$ , и это эквивалентно сдвигу на 1 бит влево, так что инструкция `MOV R0, R4, LSL#1` делает это.

`STR R0, [SP, R4, LSL#2]` записывает содержимое R0 в массив. Указатель на элемент массива вычисляется так: **SP!** указывает на начало массива, R4 это  $i$ .

Так что сдвигаем  $i$  на 2 бита влево, что эквивалентно умножению на 4 (ведь каждый элемент массива занимает 4 байта) и прибавляем это к адресу начала массива.

Во втором цикле используется обратная инструкция `LDR R2, [SP, R4, LSL#2]`. Она загружает из массива нужное значение и указатель на него вычисляется точно так же.

### Оптимизирующий Keil 6/2013 (Режим Thumb)

```

_main
    PUSH    {R4,R5,LR}
; выделить место для 20 переменных типа int + еще одной переменной:
    SUB     SP, SP, #0x54
; первый цикл:
    MOVS   R0, #0      ; i
    MOV    R5, SP      ; указатель на первый элемент массива

loc_1CE
    LSL    R1, R0, #1  ; R1=i<<1 (то же что и i*2)
    LSL    R2, R0, #2  ; R2=i<<2 (то же что и i*4)
    ADDS   R0, R0, #1  ; i=i+1
    CMP    R0, #20     ; i<20?
    STR    R1, [R5,R2] ; сохранить R1 в *(R5+R2) (то же что и R5+i*4)
    BLT   loc_1CE     ; да, i<20, запустить тело цикла снова
; второй цикл:
    MOVS   R4, #0      ; i=0
loc_1DC
    LSL    R0, R4, #2  ; R0=i<<2 (то же что и i*4)
    LDR    R2, [R5,R0] ; загрузить из *(R5+R0) (то же что и R5+i*4)
    MOVS   R1, R4
    ADR    R0, aADD    ; "a[%d]=%d\n"
    BL     __2printf
    ADDS   R4, R4, #1  ; i=i+1
    CMP    R4, #20     ; i<20?
    BLT   loc_1DC     ; да, i<20, запустить тело цикла снова
    MOVS   R0, #0      ; значение для возврата
; освободить блок в стеке, выделенное для 20-и переменных типа int
; и еще одной переменной:
    ADD    SP, SP, #0x54
    POP    {R4,R5,PC}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Код для Thumb очень похожий. В Thumb имеются отдельные инструкции для битовых сдвигов (как LSLS), вычисляющие и число для записи в массив и адрес каждого элемента массива.

Компилятор почему-то выделил в локальном стеке немного больше места, однако последние 4 байта не используются.

### Неоптимизирующий GCC 4.9.1 (ARM64)

Листинг 1.230: Неоптимизирующий GCC 4.9.1 (ARM64)

```
.LC0:
    .string "a[%d]=%d\n"
main:
; сохранить FP и LR в стековом фрейме:
    stp    x29, x30, [sp, -112]!
; установить стековый фрейм (FP=SP)
    add    x29, sp, 0
; установить начальное значение счетчика в 0
; (WZR это регистр всегда содержащий ноль):
    str    wzr, [x29,108]
; перейти на код проверки условия цикла:
    b     .L2
.L3:
; загрузить значение переменной "i":
    ldr    w0, [x29,108]
; умножить на 2:
    lsl    w2, w0, 1
; найти место массива в локальном стеке:
    add    x0, x29, 24
; загрузить 32-битное число из локального стека
; и расширить его до 64-битного учитывая знак:
    ldrsw  x1, [x29,108]
; вычислить адрес элемента (X0+X1<<2=array address+i*4) и записать W2 (i*2)
    там:
    str    w2, [x0,x1,lsr 2]
; инкремент счетчика (i):
    ldr    w0, [x29,108]
    add    w0, w0, 1
    str    w0, [x29,108]
.L2:
; проверить, закончили ли:
    ldr    w0, [x29,108]
    cmp    w0, 19
; перейти на L3 (начало тела цикла), если нет:
    ble   .L3
; здесь начинается вторая часть функции.
; устанавливаем первоначальное значение счетчика в 0.
; кстати, это же место в локальном стеке было уже использовано под счетчик
; потому что та же локальная переменная (i) используется как счетчик.
    str    wzr, [x29,108]
    b     .L4
.L5:
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; вычислить адрес массива:
    add    x0, x29, 24
; загрузить переменную "i":
    ldrsw  x1, [x29,108]
; загрузить значение из массива по адресу (X0+X1<<2 = адрес массива + i*4)
    ldr    w2, [x0,x1,ls1 2]
; загрузить адрес строки "a[%d]=%d\n":
    adrp   x0, .LC0
    add    x0, x0, :lo12:.LC0
; загрузить переменную "i" в W1 и передать её в printf() как второй аргумент:
    ldr    w1, [x29,108]
; W2 всё ещё содержит загруженный элемент из массива.
; вызов printf():
    bl     printf
; инкремент переменной "i":
    ldr    w0, [x29,108]
    add    w0, w0, 1
    str    w0, [x29,108]
.L4:
; закончили?
    ldr    w0, [x29,108]
    cmp    w0, 19
; перейти на начало тела цикла, если нет:
    ble   .L5
; возврат 0
    mov    w0, 0
; восстановить FP и LR:
    ldp    x29, x30, [sp], 112
    ret

```

## MIPS

Функция использует много S-регистров, которые должны быть сохранены. Вот почему их значения сохраняются в прологе функции и восстанавливаются в эпилоге.

Листинг 1.231: Оптимизирующий GCC 4.4.5 (IDA)

```

main:
var_70      = -0x70
var_68      = -0x68
var_14      = -0x14
var_10      = -0x10
var_C       = -0xC
var_8       = -8
var_4       = -4
; пролог функции:
    lui    $gp, (__gnu_local_gp >> 16)
    addiu  $sp, -0x80
    la     $gp, (__gnu_local_gp & 0xFFFF)
    sw     $ra, 0x80+var_4($sp)
    sw     $s3, 0x80+var_8($sp)

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

sw    $s2, 0x80+var_C($sp)
sw    $s1, 0x80+var_10($sp)
sw    $s0, 0x80+var_14($sp)
sw    $gp, 0x80+var_70($sp)
addiu $s1, $sp, 0x80+var_68
move  $v1, $s1
move  $v0, $zero
; это значение используется как терминатор цикла.
; оно было вычислено компилятором GCC на стадии компиляции:
li    $a0, 0x28 # '('

loc_34:                                # CODE XREF: main+3C
; сохранить значение в памяти:
sw    $v0, 0($v1)
; увеличивать значение (которое будет записано) на 2 на каждой итерации:
addiu $v0, 2
; дошли до терминатора цикла?
bne   $v0, $a0, loc_34
; в любом случае, добавляем 4 к адресу:
addiu $v1, 4
; цикл заполнения массива закончился
; начало второго цикла
la    $s3, $LC0 # "a[%d]=%d\n"
; переменная "i" будет находиться в $s0:
move  $s0, $zero
li    $s2, 0x14

loc_54:                                # CODE XREF: main+70
; вызов printf():
lw    $t9, (printf & 0xFFFF)($gp)
lw    $a2, 0($s1)
move  $a1, $s0
move  $a0, $s3
jalr  $t9
; инкремент "i":
addiu $s0, 1
lw    $gp, 0x80+var_70($sp)
; перейти на начало тела цикла, если конец еще не достигнут:
bne   $s0, $s2, loc_54
; передвинуть указатель на следующее 32-битное слово:
addiu $s1, 4
; эпилог функции
lw    $ra, 0x80+var_4($sp)
move  $v0, $zero
lw    $s3, 0x80+var_8($sp)
lw    $s2, 0x80+var_C($sp)
lw    $s1, 0x80+var_10($sp)
lw    $s0, 0x80+var_14($sp)
jr    $ra
addiu $sp, 0x80

$LC0: .ascii "a[%d]=%d\n"<0> # DATA XREF: main+44

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Интересная вещь: здесь два цикла и в первом не нужна переменная  $i$ , а нужна только переменная  $i * 2$  (скачущая через 2 на каждой итерации) и ещё адрес в памяти (скачущий через 4 на каждой итерации).

Так что мы видим здесь две переменных: одна (в  $\$V0$ ) увеличивается на 2 каждый раз, и вторая (в  $\$V1$ ) — на 4.

Второй цикл содержит вызов `printf()`. Он должен показывать значение  $i$  пользователю, поэтому здесь есть переменная, увеличивающаяся на 1 каждый раз (в  $\$S0$ ), а также адрес в памяти (в  $\$S1$ ) увеличивающийся на 4 каждый раз.

Это напоминает нам оптимизацию циклов: ?? (стр. ??). Цель оптимизации в том, чтобы избавиться от операций умножения.

## 1.26.2. Переполнение буфера

### Чтение за пределами массива

Итак, индексация массива — это просто *массив[индекс]*. Если вы присмотритесь к коду, в цикле печати значений массива через `printf()` вы не увидите проверок индекса, *меньше ли он двадцати?* А что будет если он будет 20 или больше? Эта одна из особенностей Си/Си++, за которую их, собственно, и ругают.

Вот код, который и компилируется и работает:

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<20; i++)
        a[i]=i*2;

    printf ("a[20]=%d\n", a[20]);

    return 0;
};
```

Вот результат компиляции в (MSVC 2008):

Листинг 1.232: Неоптимизирующий MSVC 2008

```
$SG2474 DB    'a[20]=%d', 0aH, 00H

_i$ = -84 ; size = 4
_a$ = -80 ; size = 80
_main PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 84
    mov     DWORD PTR _i$[ebp], 0
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

    jmp     SHORT $LN3@main
$LN2@main:
    mov     eax, DWORD PTR _i$[ebp]
    add     eax, 1
    mov     DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp     DWORD PTR _i$[ebp], 20
    jge     SHORT $LN1@main
    mov     ecx, DWORD PTR _i$[ebp]
    shl     ecx, 1
    mov     edx, DWORD PTR _i$[ebp]
    mov     DWORD PTR _a$[ebp+edx*4], ecx
    jmp     SHORT $LN2@main
$LN1@main:
    mov     eax, DWORD PTR _a$[ebp+80]
    push   eax
    push   OFFSET $SG2474 ; 'a[20]=%d'
    call   DWORD PTR __imp__printf
    add     esp, 8
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main     ENDP
_TEXT    ENDS
END

```

Данный код при запуске выдал вот такой результат:

Листинг 1.233: OllyDbg: вывод в консоль

```
a[20]=1638280
```

Это просто *что-то*, что волею случая лежало в стеке рядом с массивом, через 80 байт от его первого элемента.

Попробуем узнать в OllyDbg, что это за значение. Загружаем и находим это значение, находящееся точно после последнего элемента массива:

The screenshot shows the CPU window with assembly code. The instruction at address 0040102C is `MOV EAX, DWORD PTR SS:[LOCAL.0]`. The stack window shows the return address `0018FF44=0018FF88`. The registers window shows `EAX: 00000014`, `ECX: 00000026`, `EDX: 00000013`, and `EIP: 0040102C`.

Рис. 1.88: OllyDbg: чтение 20-го элемента и вызов printf()

Что это за значение? Судя по разметке стека, это сохраненное значение регистра EBP.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Трассируем далее, и видим, как оно восстанавливается:

The screenshot shows the CPU window of OllyDbg. The assembly code is as follows:

```

00401000 55          PUSH EBP
00401001 8BEC       MOV EBP,ESP
00401003 83EC 54    SUB ESP,54
00401006 C745 AC 0000 MOV DWORD PTR SS:[LOCAL.21],0
0040100D EB 09     JMP SHORT 00401018
0040100F 8B45 AC    MOV EAX,DWORD PTR SS:[LOCAL.21]
00401012 83C0 01    ADD EAX,1
00401015 8945 AC    MOV DWORD PTR SS:[LOCAL.21],EAX
00401018 837D AC 14 CMP DWORD PTR SS:[LOCAL.21],14
0040101C 7D 0E     JGE SHORT 0040102C
0040101E 8B4D AC    MOV ECX,DWORD PTR SS:[LOCAL.21]
00401021 D1E1     SHL ECX,1
00401023 8B55 AC    MOV EDX,DWORD PTR SS:[LOCAL.21]
00401026 894C 95 B0 MOV DWORD PTR SS:[EDX*4+EBP-50],ECX
0040102A EB E3     JMP SHORT 0040100F
0040102C 8B45 00    MOV EAX,DWORD PTR SS:[LOCAL.0]
0040102F 50       PUSH EAX
00401030 68 00304000 PUSH OFFSET 00403000
00401035 FF15 A0204000 CALL DWORD PTR DS:[&MSVCRC90.printf]
0040103B 83C4 08    ADD ESP,8
0040103E 33C0     XOR EAX,EAX
00401040 8BE5     MOV ESP,EBP
00401043 5D       POP EBP
00401043 C3       RETN
00401044 68 28144000 PUSH 00401428
00401049 E8 9D030000 CALL 004013EB
0040104E A1 50304000 MOV EAX,DWORD PTR DS:[403050]
00401053 C742 24 3C304000 MOV DWORD PTR SS:[LOCAL.01],OFFSET 00403C30
  
```

The Registers (FPU) window shows:

```

EAX 00000000
ECX 7555215C MSVCRC90.7555215C
EDX 00000000
EBX 00000000
ESP 0018FF48
EBP 0018FF88
ESI 00000001
EDI 0040338C r.0040338C
EIP 00401043 r.00401043
  
```

The stack dump at the bottom shows the return address 0018FF48:

```

Address Hex dump ASCII (ANSI)
0018FF44 00 00 00 00 02 00 00 00 04 00 00 00 06 00 00 00 0
0018FF48 03 00 00 00 0A 00 00 00 0C 00 00 00 0E 00 00 00 0
0018FF14 10 00 00 00 12 00 00 00 14 00 00 00 16 00 00 00 0
0018FF24 18 00 00 00 1A 00 00 00 1C 00 00 00 1E 00 00 00 0
0018FF34 20 00 00 00 22 00 00 00 24 00 00 00 26 00 00 00 0
0018FF44 88 FF 18 00 9E 11 40 00 01 00 00 00 A0 4D 6E 00 0
0018FF54 48 28 6E 00 2F 0C 91 02 00 00 00 00 00 00 00 00 0
0018FF64 00 E9 FD 7E 00 00 00 00 00 00 00 53 FF 18 00 00 0
0018FF74 2C 2A 03 72 C4 FF 18 00 15 16 40 00 9F D2 C9 02 0
0018FF84 00 00 00 00 94 FF 18 00 8A 33 F6 76 00 E0 FD 7E 0
0018FF94 04 FF 18 00 72 9F D3 77 00 E0 FD 7E 78 22 89 7E 0
0018FFA4 00 00 00 00 00 00 00 00 00 E0 FD 7E 00 00 00 00 0
0018FFB4 00 00 00 00 00 00 00 00 00 FF 18 00 00 00 00 00 0
0018FFC4 E4 FF 18 00 F5 71 D7 77 FC 18 43 09 00 00 00 00 0
0018FFD4 EC FF 18 00 45 9F D3 77 E6 12 40 00 00 E9 FD 7E 0
0018FFE4 FF FF FF FF 23 74 D3 77 00 00 00 00 00 00 00 0
0018FFF4 E6 12 40 00 00 E0 FD 7E 00 00 00 00 00 00 00 0
  
```

Рис. 1.89: OllyDbg: восстановление EBP

Действительно, а как могло бы быть иначе? Компилятор мог бы встроить какой-то код, каждый раз проверяющий индекс на соответствие пределам массива, как в языках программирования более высокого уровня<sup>124</sup>, что делало бы запускаяемый код медленнее.

### Запись за пределы массива

Итак, мы прочитали какое-то число из стека явно *нелегально*, а что если мы запишем?

Вот что мы пишем:

```
#include <stdio.h>
```

```
int main()
```

<sup>124</sup>Java, Python, и т. д.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
{
    int a[20];
    int i;

    for (i=0; i<30; i++)
        a[i]=i;

    return 0;
};
```

## MSVC

И вот что имеем на ассемблере:

Листинг 1.234: Неоптимизирующий MSVC 2008

```
_TEXT    SEGMENT
_i$ = -84 ; size = 4
_a$ = -80 ; size = 80
_main    PROC
push     ebp
mov      ebp, esp
sub      esp, 84
mov      DWORD PTR _i$[ebp], 0
jmp      SHORT $LN3@main
$LN2@main:
mov      eax, DWORD PTR _i$[ebp]
add      eax, 1
mov      DWORD PTR _i$[ebp], eax
$LN3@main:
cmp      DWORD PTR _i$[ebp], 30 ; 0000001eH
jge      SHORT $LN1@main
mov      ecx, DWORD PTR _i$[ebp]
mov      edx, DWORD PTR _i$[ebp] ; явный промах компилятора. эта
инструкция лишняя.
mov      DWORD PTR _a$[ebp+ecx*4], edx ; а здесь в качестве второго операнда
подошел бы ECX.
jmp      SHORT $LN2@main
$LN1@main:
xor      eax, eax
mov      esp, ebp
pop      ebp
ret      0
_main    ENDP
```

Запускаете скомпилированную программу, и она падает. Немудрено. Но давайте теперь узнаем, где именно.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Загружаем в OllyDbg, трассируем пока запишутся все 30 элементов:

The screenshot displays the OllyDbg interface with the following components:

- Assembly View:** Shows assembly instructions from address 00401000 to 0040102E. The instruction at 0040102F is `CALL 004013D7`, which is highlighted in red. The instruction at 00401030 is `MOV EDX, DWORD PTR DS:[403040]`.
- Registers (FPU):** Shows the state of various registers. The `EBP` register is highlighted in red and contains the value `00000014`. Other registers like `EAX`, `EAX`, `ECX`, `EDX`, `EBX`, `ESP`, `ESI`, `EDI`, and `EIP` are also visible.
- Memory Dump:** Shows a stack of memory starting at address `0018FF48`. The dump contains 30 elements, all of which are `00` (zero). The address `0018FF48` is highlighted in red.

Рис. 1.90: OllyDbg: после восстановления EBP

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Доходим до конца функции:

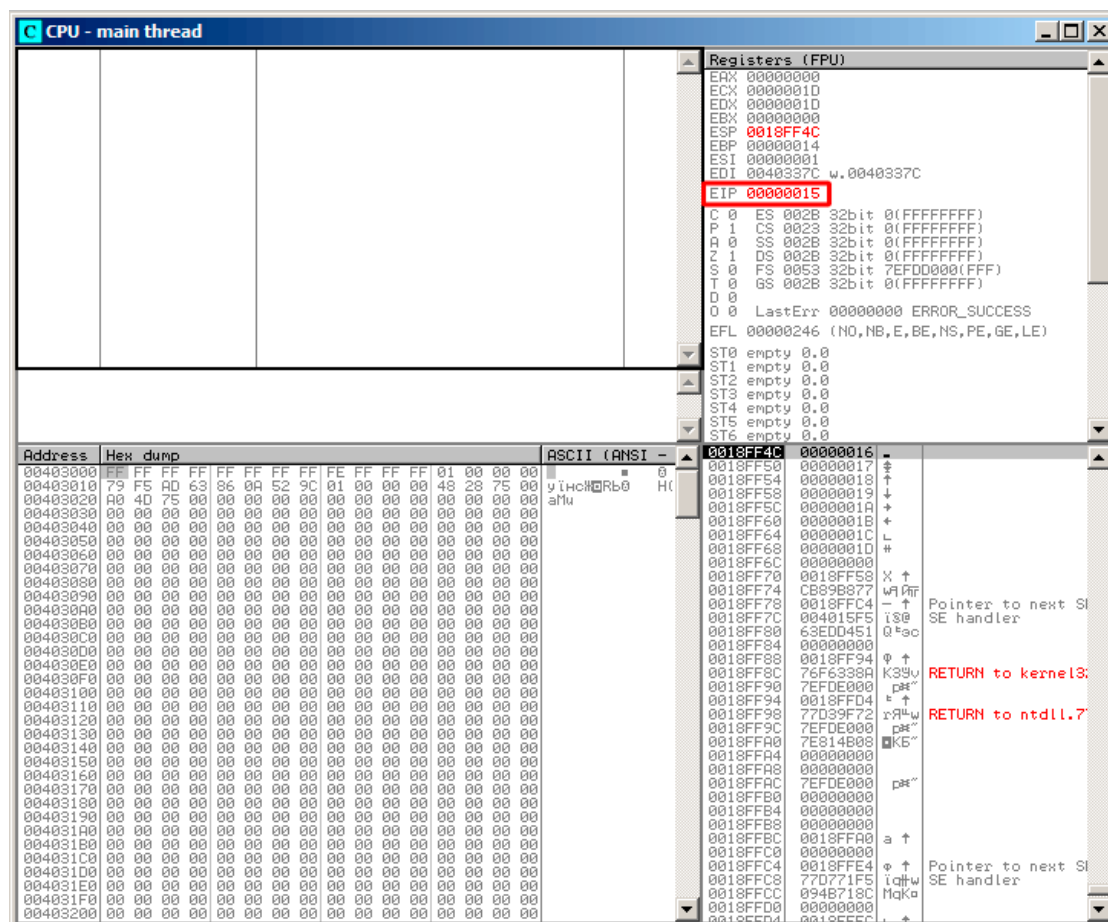


Рис. 1.91: OllYDbg: EIP восстановлен, но OllYDbg не может дизассемблировать по адресу 0x15

Итак, следите внимательно за регистрами.

EIP теперь 0x15. Это явно нелегальный адрес для кода — по крайней мере, win32-кода! Мы там как-то очутились, причем, сами того не хотели. Интересен также тот факт, что в EBP хранится 0x14, а в ECX и EDX хранится 0x1D.

Ещё немного изучим разметку стека.

После того как управление передалось в main(), в стек было сохранено значение EBP. Затем для массива и переменной *i* было выделено 84 байта. Это  $(20+1) * \text{sizeof}(\text{int})$ . ESP сейчас указывает на переменную *\_i* в локальном стеке и при исполнении следующего PUSH что-либо, вот это вот что-либо появится рядом с *\_i*.

Вот так выглядит разметка стека пока управление находится внутри main():

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

ESP	4 байта выделенных для переменной <i>i</i>
ESP+4	80 байт выделенных для массива <i>a[20]</i>
ESP+84	сохраненное значение EBP
ESP+88	адрес возврата

Выражение `a[19]=что_нибудь` записывает последний *int* в пределах массива (пока что в пределах!).

Выражение `a[20]=что_нибудь` записывает *что\_нибудь* на место где сохранено значение EBP.

Обратите внимание на состояние регистров на момент падения процесса. В нашем случае в 20-й элемент записалось значение 20. И вот всё дело в том, что заканчиваясь, эпилог функции восстанавливал значение EBP (20 в десятичной системе это как раз 0x14 в шестнадцатеричной). Далее выполнялась инструкция RET, которая на самом деле эквивалентна POP EIP.

Инструкция RET вытащила из стека адрес возврата (это адрес где-то внутри **CRT!**, которая вызвала `main()`), а там было записано 21 в десятичной системе, то есть 0x15 в шестнадцатеричной. И вот процессор оказался по адресу 0x15, но исполняемого кода там нет, так что случилось исключение.

Добро пожаловать! Это называется *buffer overflow*<sup>125</sup>.

Замените массив *int* на строку (массив *char*), нарочно создайте слишком длинную строку, передайте её в ту программу, в ту функцию, которая не проверяя длину строки копирует её в слишком короткий буфер, и вы сможете указать программе, по какому именно адресу перейти. Не всё так просто в реальности, конечно, но началось всё с этого. Классическая статья об этом: [Aleph One, *Smashing The Stack For Fun And Profit*, (1996)]<sup>126</sup>.

## GCC

Попробуем то же самое в GCC 4.4.1. У нас выходит такое:

```

main                public main
                   proc near
a                   = dword ptr -54h
i                   = dword ptr -4

                   push    ebp
                   mov     ebp, esp
                   sub     esp, 60h ; 96
                   mov     [ebp+i], 0
                   jmp     short loc_80483D1
loc_80483C3:
                   mov     eax, [ebp+i]
                   mov     edx, [ebp+i]
                   mov     [ebp+eax*4+a], edx

```

<sup>125</sup>[wikipedia](#)

<sup>126</sup>Также доступно здесь: <http://yurichev.com/mirrors/phrack/p49-0x0e.txt>

```

loc_80483D1:  add    [ebp+i], 1
             cmp    [ebp+i], 1Dh
             jle   short loc_80483C3
             mov   eax, 0
             leave
             retn
main         endp

```

Запуск этого в Linux выдаст: Segmentation fault.

Если запустить полученное в отладчике GDB, получим:

```

(gdb) r
Starting program: /home/dennis/RE/1

Program received signal SIGSEGV, Segmentation fault.
0x00000016 in ?? ()
(gdb) info registers
eax            0x0          0
ecx            0xd2f96388   -755407992
edx            0x1d          29
ebx            0x26eff4    2551796
esp            0xbffff4b0   0xbffff4b0
ebp            0x15          0x15
esi            0x0          0
edi            0x0          0
eip            0x16          0x16
eflags        0x10202    [ IF RF ]
cs             0x73          115
ss             0x7b          123
ds             0x7b          123
es             0x7b          123
fs             0x0          0
gs             0x33          51
(gdb)

```

Значения регистров немного другие, чем в примере win32, потому что разметка стека чуть другая.

### 1.26.3. Защита от переполнения буфера

В наше время пытаются бороться с переполнением буфера невзирая на халатность программистов на Си/Си++. В MSVC есть опции вроде<sup>127</sup>:

```

/RTCs Stack Frame runtime checking
/GZ Enable stack checks (/RTCs)

```

<sup>127</sup>описания защит, которые компилятор может вставлять в код: [wikipedia.org/wiki/Buffer\\_overflow\\_protection](http://wikipedia.org/wiki/Buffer_overflow_protection)



Одним из методов является вставка в прологе функции некоего случайного значения в область локальных переменных и проверка этого значения в эпилоге функции перед выходом. Если проверка не прошла, то не выполнять инструкцию RET, а остановиться (или зависнуть). Процесс зависнет, но это лучше, чем удаленная атака на ваш компьютер.

Это случайное значение иногда называют «канарейкой»<sup>128</sup>, по аналогии с шахтной канарейкой<sup>129</sup>. Раньше использовали шахтеры, чтобы определять, есть ли в шахте опасный газ.

Канарейки очень к нему чувствительны и либо проявляли сильное беспокойство, либо гибли от газа.

Если скомпилировать наш простейший пример работы с массивом (?? (стр. ??)) в **MSVC!** с опцией RTC1 или RTCs, в конце нашей функции будет вызов функции @ \_RTC\_CheckStackVars@8, проверяющей корректность «канарейки».

Посмотрим, как дела обстоят в GCC. Возьмем пример из секции про alloca() (?? (стр. ??)):

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

По умолчанию, без дополнительных ключей, GCC 4.7.3 вставит в код проверку «канарейки»:

Листинг 1.235: GCC 4.7.3

```
.LC0:
    .string "hi! %d, %d, %d\n"
f:
    push    ebp
    mov     ebp, esp
    push    ebx
    sub     esp, 676
    lea    ebx, [esp+39]
```

<sup>128</sup> «canary» в англоязычной литературе

<sup>129</sup> [miningwiki.ru/wiki/Канарейка\\_в\\_шахте](http://miningwiki.ru/wiki/Канарейка_в_шахте)

```

and    ebx, -16
mov    DWORD PTR [esp+20], 3
mov    DWORD PTR [esp+16], 2
mov    DWORD PTR [esp+12], 1
mov    DWORD PTR [esp+8], OFFSET FLAT:.LC0 ; "hi! %d, %d, %d\n"
mov    DWORD PTR [esp+4], 600
mov    DWORD PTR [esp], ebx
mov    eax, DWORD PTR gs:20 ; канарейка
mov    DWORD PTR [ebp-12], eax
xor    eax, eax
call   _snprintf
mov    DWORD PTR [esp], ebx
call   puts
mov    eax, DWORD PTR [ebp-12]
xor    eax, DWORD PTR gs:20 ; проверка канарейки
jne    .L5
mov    ebx, DWORD PTR [ebp-4]
leave
ret
.L5:
call   __stack_chk_fail

```

Случайное значение находится в `gs:20`. Оно записывается в стек, затем, в конце функции, значение в стеке сравнивается с корректной «канарейкой» в `gs:20`. Если значения не равны, будет вызвана функция `__stack_chk_fail` и в консоли мы увидим что-то вроде такого (Ubuntu 13.04 x86):

```

*** buffer overflow detected ***: ./2_1 terminated
===== Backtrace: =====
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x63)[0xb7699bc3]
/lib/i386-linux-gnu/libc.so.6(+0x10593a)[0xb769893a]
/lib/i386-linux-gnu/libc.so.6(+0x105008)[0xb7698008]
/lib/i386-linux-gnu/libc.so.6(_IO_default_xsputn+0x8c)[0xb7606e5c]
/lib/i386-linux-gnu/libc.so.6(_IO_vfprintf+0x165)[0xb75d7a45]
/lib/i386-linux-gnu/libc.so.6(__vsprintf_chk+0xc9)[0xb76980d9]
/lib/i386-linux-gnu/libc.so.6(__sprintf_chk+0x2f)[0xb7697fef]
./2_1[0x8048404]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf5)[0xb75ac935]
===== Memory map: =====
08048000-08049000 r-xp 00000000 08:01 2097586 /home/dennis/2_1
08049000-0804a000 r--p 00000000 08:01 2097586 /home/dennis/2_1
0804a000-0804b000 rw-p 00001000 08:01 2097586 /home/dennis/2_1
094d1000-094f2000 rw-p 00000000 00:00 0 [heap]
b7560000-b757b000 r-xp 00000000 08:01 1048602 /lib/i386-linux-gnu/
↳ libgcc_s.so.1
b757b000-b757c000 r--p 0001a000 08:01 1048602 /lib/i386-linux-gnu/
↳ libgcc_s.so.1
b757c000-b757d000 rw-p 0001b000 08:01 1048602 /lib/i386-linux-gnu/
↳ libgcc_s.so.1
b7592000-b7593000 rw-p 00000000 00:00 0
b7593000-b7740000 r-xp 00000000 08:01 1050781 /lib/i386-linux-gnu/libc
↳ -2.17.so
b7740000-b7742000 r--p 001ad000 08:01 1050781 /lib/i386-linux-gnu/libc

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

↳ -2.17.so
b7742000-b7743000 rw-p 001af000 08:01 1050781 /lib/i386-linux-gnu/libc.so.2
↳ -2.17.so
b7743000-b7746000 rw-p 00000000 00:00 0
b775a000-b775d000 rw-p 00000000 00:00 0
b775d000-b775e000 r-xp 00000000 00:00 0 [vdso]
b775e000-b777e000 r-xp 00000000 08:01 1050794 /lib/i386-linux-gnu/ld.so.2
↳ -2.17.so
b777e000-b777f000 r--p 0001f000 08:01 1050794 /lib/i386-linux-gnu/ld.so.2
↳ -2.17.so
b777f000-b7780000 rw-p 00020000 08:01 1050794 /lib/i386-linux-gnu/ld.so.2
↳ -2.17.so
bff35000-bff56000 rw-p 00000000 00:00 0 [stack]
Aborted (core dumped)

```

gs это так называемый сегментный регистр. Эти регистры широко использовались во времена MS-DOS и DOS-экстендеров. Сейчас их функция немного изменилась. Если говорить кратко, в Linux gs всегда указывает на **TLS!**<sup>130</sup> (?? (стр. ??)) — там находится различная информация, специфичная для выполняющегося потока.

Кстати, в win32 эту же роль играет сегментный регистр fs, он всегда указывает на **TIB!**<sup>131 132</sup>.

Больше информации можно почерпнуть из исходных кодов Linux (по крайней мере, в версии 3.11):

в файле `arch/x86/include/asm/stackprotector.h` в комментариях описывается эта переменная.

### Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

Возвращаясь к нашему простому примеру (?? (стр. ??)), можно посмотреть, как LLVM добавит проверку «канарейки»:

```

_main
var_64      = -0x64
var_60      = -0x60
var_5C      = -0x5C
var_58      = -0x58
var_54      = -0x54
var_50      = -0x50
var_4C      = -0x4C
var_48      = -0x48
var_44      = -0x44
var_40      = -0x40
var_3C      = -0x3C
var_38      = -0x38
var_34      = -0x34

```

<sup>130</sup>**TLS!**

<sup>131</sup>**TIB!**

<sup>132</sup>[wikipedia.org/wiki/Win32\\_Thread\\_Information\\_Block](http://wikipedia.org/wiki/Win32_Thread_Information_Block)

```

var_30      = -0x30
var_2C      = -0x2C
var_28      = -0x28
var_24      = -0x24
var_20      = -0x20
var_1C      = -0x1C
var_18      = -0x18
canary      = -0x14
var_10      = -0x10

    PUSH      {R4-R7,LR}
    ADD      R7, SP, #0xC
    STR.W    R8, [SP,#0xC+var_10]!
    SUB      SP, SP, #0x54
    MOVW     R0, #a0bjc_methtype ; "objc_methtype"
    MOVS     R2, #0
    MOVT.W   R0, #0
    MOVS     R5, #0
    ADD      R0, PC
    LDR.W    R8, [R0]
    LDR.W    R0, [R8]
    STR      R0, [SP,#0x64+canary]
    MOVS     R0, #2
    STR      R2, [SP,#0x64+var_64]
    STR      R0, [SP,#0x64+var_60]
    MOVS     R0, #4
    STR      R0, [SP,#0x64+var_5C]
    MOVS     R0, #6
    STR      R0, [SP,#0x64+var_58]
    MOVS     R0, #8
    STR      R0, [SP,#0x64+var_54]
    MOVS     R0, #0xA
    STR      R0, [SP,#0x64+var_50]
    MOVS     R0, #0xC
    STR      R0, [SP,#0x64+var_4C]
    MOVS     R0, #0xE
    STR      R0, [SP,#0x64+var_48]
    MOVS     R0, #0x10
    STR      R0, [SP,#0x64+var_44]
    MOVS     R0, #0x12
    STR      R0, [SP,#0x64+var_40]
    MOVS     R0, #0x14
    STR      R0, [SP,#0x64+var_3C]
    MOVS     R0, #0x16
    STR      R0, [SP,#0x64+var_38]
    MOVS     R0, #0x18
    STR      R0, [SP,#0x64+var_34]
    MOVS     R0, #0x1A
    STR      R0, [SP,#0x64+var_30]
    MOVS     R0, #0x1C
    STR      R0, [SP,#0x64+var_2C]
    MOVS     R0, #0x1E
    STR      R0, [SP,#0x64+var_28]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

MOVS    R0, #0x20
STR     R0, [SP,#0x64+var_24]
MOVS    R0, #0x22
STR     R0, [SP,#0x64+var_20]
MOVS    R0, #0x24
STR     R0, [SP,#0x64+var_1C]
MOVS    R0, #0x26
STR     R0, [SP,#0x64+var_18]
MOV     R4, 0xFDA ; "a[%d]=%d\n"
MOV     R0, SP
ADDS   R6, R0, #4
ADD     R4, PC
B       loc_2F1C

; начало второго цикла

loc_2F14
ADDS   R0, R5, #1
LDR.W  R2, [R6,R5,LSL#2]
MOV     R5, R0

loc_2F1C
MOV     R0, R4
MOV     R1, R5
BLX    _printf
CMP     R5, #0x13
BNE    loc_2F14
LDR.W  R0, [R8]
LDR     R1, [SP,#0x64+canary]
CMP     R0, R1
ITTTT EQ ; канарейка все еще верна?
MOVEQ  R0, #0
ADDEQ  SP, SP, #0x54
LDREQ.W R8, [SP+0x64+var_64],#4
POPEQ  {R4-R7,PC}
BLX    __stack_chk_fail

```

Во-первых, LLVM «развернул» цикл и все значения записываются в массив по одному, уже вычисленные, потому что LLVM посчитал что так будет быстрее.

Кстати, инструкции режима ARM позволяют сделать это ещё быстрее и это может быть вашим домашним заданием.

В конце функции мы видим сравнение «канареек» — той что лежит в локальном стеке и корректной, на которую ссылается регистр R8.

Если они равны, срабатывает блок из четырех инструкций при помощи ITTTT EQ. Это запись 0 в R0, эпилог функции и выход из нее.

Если «канарейки» не равны, блок не срабатывает и происходит переход на функцию `__stack_chk_fail`, которая остановит работу программы.

### 1.26.4. Еще немного о массивах

Теперь понятно, почему нельзя написать в исходном коде на Си/Си++ что-то вроде:

```
void f(int size)
{
    int a[size];
    ...
};
```

Чтобы выделить место под массив в локальном стеке, компилятору нужно знать размер массива, чего он на стадии компиляции, разумеется, знать не может.

Если вам нужен массив произвольной длины, то выделите столько, сколько нужно, через `malloc()`, а затем обращайтесь к выделенному блоку байт как к массиву того типа, который вам нужен.

Либо используйте возможность стандарта C99 [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007)6.7.5/2], и внутри это очень похоже на `alloca()` (?? (стр. ??)).

Для работы с памятью, можно также воспользоваться библиотекой сборщика мусора в Си.

А для языка Си++ есть библиотеки с поддержкой умных указателей.

### 1.26.5. Массив указателей на строки

Вот пример массива указателей.

Листинг 1.236: Получить имя месяца

```
#include <stdio.h>

const char* month1[]=
{
    "January", "February", "March", "April",
    "May", "June", "July", "August",
    "September", "October", "November", "December"
};

// в пределах 0..11
const char* get_month1 (int month)
{
    return month1[month];
};
```

## x64

Листинг 1.237: Оптимизирующий MSVC 2013 x64

_DATA	SEGMENT	
month1	DQ	FLAT:\$SG3122
	DQ	FLAT:\$SG3123

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

DQ    FLAT:$SG3124
DQ    FLAT:$SG3125
DQ    FLAT:$SG3126
DQ    FLAT:$SG3127
DQ    FLAT:$SG3128
DQ    FLAT:$SG3129
DQ    FLAT:$SG3130
DQ    FLAT:$SG3131
DQ    FLAT:$SG3132
DQ    FLAT:$SG3133
$SG3122 DB 'January', 00H
$SG3123 DB 'February', 00H
$SG3124 DB 'March', 00H
$SG3125 DB 'April', 00H
$SG3126 DB 'May', 00H
$SG3127 DB 'June', 00H
$SG3128 DB 'July', 00H
$SG3129 DB 'August', 00H
$SG3130 DB 'September', 00H
$SG3156 DB '%s', 0aH, 00H
$SG3131 DB 'October', 00H
$SG3132 DB 'November', 00H
$SG3133 DB 'December', 00H
_DATA  ENDS

month$ = 8
get_month1 PROC
    movsxd  rax, ecx
    lea    rcx, OFFSET FLAT:month1
    mov    rax, QWORD PTR [rcx+rax*8]
    ret    0
get_month1 ENDP

```

Код очень простой:

- Первая инструкция MOVSD копирует 32-битное значение из ECX (где передается аргумент *month*) в RAX со знаковым расширением (потому что аргумент *month* имеет тип *int*).

Причина расширения в том, что это значение будет использоваться в вычислениях наряду с другими 64-битными значениями.

Таким образом, оно должно быть расширено до 64-битного <sup>133</sup>.

- Затем адрес таблицы указателей загружается в RCX.
- В конце концов, входное значение (*month*) умножается на 8 и прибавляется к адресу. Действительно: мы в 64-битной среде и все адреса (или указатели) требуют для хранения именно 64 бита (или 8 байт). Следовательно, каждый элемент таблицы имеет ширину в 8 байт. Вот почему для

<sup>133</sup>Это немного странная вещь, но отрицательный индекс массива может быть передан как *month* (отрицательные индексы массивов будут рассмотрены позже: ?? (стр. ??)). И если так будет, отрицательное значение типа *int* будет расширено со знаком корректно и соответствующий элемент перед таблицей будет выбран. Всё это не будет корректно работать без знакового расширения.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

выбора элемента под нужным номером нужно пропустить  $month * 8$  байт от начала. Это то, что делает MOV. Эта инструкция также загружает элемент по этому адресу. Для 1, элемент будет указателем на строку, содержащую «February», и т. д.

Оптимизирующий GCC 4.9 может это сделать даже лучше <sup>134</sup>:

Листинг 1.238: Оптимизирующий GCC 4.9 x64

```
movsx rdi, edi
mov   rax, QWORD PTR month1[0+rdi*8]
ret
```

## 32-bit MSVC

Скомпилируем также в 32-битном компиляторе MSVC:

Листинг 1.239: Оптимизирующий MSVC 2013 x86

```
_month$ = 8
_get_month1 PROC
    mov     eax, DWORD PTR _month$[esp-4]
    mov     eax, DWORD PTR _month1[eax*4]
    ret     0
_get_month1 ENDP
```

Входное значение не нужно расширять до 64-битного значения, так что оно используется как есть.

И оно умножается на 4, потому что элементы таблицы имеют ширину 32 бита или 4 байта.

## 32-битный ARM

### ARM в режиме ARM

Листинг 1.240: Оптимизирующий Keil 6/2013 (Режим ARM)

```
get_month1 PROC
    LDR    r1, |L0.100|
    LDR    r0, [r1, r0, LSL #2]
    BX    lr
    ENDP

|L0.100|
    DCD    |.data|

    DCB    "January", 0
    DCB    "February", 0
```

<sup>134</sup>В листинге осталось «0+», потому что вывод ассемблера GCC не так скрупулёзен, чтобы убрать это. Это *displacement* и он здесь нулевой.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

DCB      "March",0
DCB      "April",0
DCB      "May",0
DCB      "June",0
DCB      "July",0
DCB      "August",0
DCB      "September",0
DCB      "October",0
DCB      "November",0
DCB      "December",0

AREA ||.data||, DATA, ALIGN=2
month1
DCD      ||.conststring||
DCD      ||.conststring||+0x8
DCD      ||.conststring||+0x11
DCD      ||.conststring||+0x17
DCD      ||.conststring||+0x1d
DCD      ||.conststring||+0x21
DCD      ||.conststring||+0x26
DCD      ||.conststring||+0x2b
DCD      ||.conststring||+0x32
DCD      ||.conststring||+0x3c
DCD      ||.conststring||+0x44
DCD      ||.conststring||+0x4d

```

Адрес таблицы загружается в R1.

Всё остальное делается, используя только одну инструкцию LDR.

Входное значение *month* сдвигается влево на 2 (что тоже самое что и умножение на 4), это значение прибавляется к R1 (где находится адрес таблицы) и затем элемент таблицы загружается по этому адресу.

32-битный элемент таблицы загружается в R0 из таблицы.

### ARM в режиме Thumb

Код почти такой же, только менее плотный, потому что здесь, в инструкции LDR, нельзя задать суффикс LSL:

```

get_month1 PROC
    LSLS    r0,r0,#2
    LDR     r1,|L0.64|
    LDR     r0,[r1,r0]
    BX     lr
    ENDP

```

### ARM64

#### Листинг 1.241: Оптимизирующий GCC 4.9 ARM64

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

get_month1:
    adrp    x1, .LANCHOR0
    add     x1, x1, :lo12:.LANCHOR0
    ldr     x0, [x1,w0,sxtw 3]
    ret

.LANCHOR0 = . + 0
    .type   month1, %object
    .size   month1, 96
month1:
    .xword  .LC2
    .xword  .LC3
    .xword  .LC4
    .xword  .LC5
    .xword  .LC6
    .xword  .LC7
    .xword  .LC8
    .xword  .LC9
    .xword  .LC10
    .xword  .LC11
    .xword  .LC12
    .xword  .LC13

.LC2:
    .string "January"
.LC3:
    .string "February"
.LC4:
    .string "March"
.LC5:
    .string "April"
.LC6:
    .string "May"
.LC7:
    .string "June"
.LC8:
    .string "July"
.LC9:
    .string "August"
.LC10:
    .string "September"
.LC11:
    .string "October"
.LC12:
    .string "November"
.LC13:
    .string "December"

```

Адрес таблицы загружается в X1 используя пару ADRP/ADD.

Соответствующий элемент выбирается используя одну инструкцию LDR, которая берет W0 (регистр, где находится значение входного аргумента *month*), сдвигает его на 3 бита влево (что то же самое что и умножение на 8), расширяет его, учитывая знак (это то, что означает суффикс «sxtw») и прибавляет к

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

X0.

Затем 64-битное значение загружается из таблицы в X0.

## MIPS

Листинг 1.242: Оптимизирующий GCC 4.4.5 (IDA)

```

get_month1:
; загрузить адрес таблицы в $v0:
    la    $v0, month1
; взять входное значение и умножить его на 4:
    sll   $a0, 2
; сложить адрес таблицы и умноженное значение:
    addu  $a0, $v0
; загрузить элемент таблицы по этому адресу в $v0:
    lw    $v0, 0($a0)
; возврат
    jr    $ra
    or    $at, $zero ; branch delay slot, NOP

    .data # .data.rel.local
    .globl month1
month1:
    .word aJanuary      # "January"
    .word aFebruary     # "February"
    .word aMarch        # "March"
    .word aApril        # "April"
    .word aMay          # "May"
    .word aJune         # "June"
    .word aJuly         # "July"
    .word aAugust       # "August"
    .word aSeptember   # "September"
    .word aOctober      # "October"
    .word aNovember     # "November"
    .word aDecember     # "December"

    .data # .rodata.str1.4
aJanuary:
    .ascii "January"<0>
aFebruary:
    .ascii "February"<0>
aMarch:
    .ascii "March"<0>
aApril:
    .ascii "April"<0>
aMay:
    .ascii "May"<0>
aJune:
    .ascii "June"<0>
aJuly:
    .ascii "July"<0>
aAugust:
    .ascii "August"<0>
aSeptember:
    .ascii "September"<0>
aOctober:
    .ascii "October"<0>
aNovember:
    .ascii "November"<0>
aDecember:
    .ascii "December"<0>

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

## Переполнение массива

Наша функция принимает значения в пределах 0..11, но что будет, если будет передано 12?

В таблице в этом месте нет элемента. Так что функция загрузит какое-то значение, которое волею случая находится там, и вернет его.

Позже, какая-то другая функция попытается прочитать текстовую строку по этому адресу и, возможно, упадет.

Скомпилируем этот пример в MSVC для win64 и откроем его в **IDA!** чтобы посмотреть, что линкер расположил после таблицы:

Листинг 1.243: Исполняемый файл в IDA

```

off_140011000  dq offset aJanuary_1      ; DATA XREF: .text:0000000140001003
                ; "January"
                dq offset aFebruary_1    ; "February"
                dq offset aMarch_1       ; "March"
                dq offset aApril_1      ; "April"
                dq offset aMay_1        ; "May"
                dq offset aJune_1       ; "June"
                dq offset aJuly_1       ; "July"
                dq offset aAugust_1     ; "August"
                dq offset aSeptember_1  ; "September"
                dq offset aOctober_1    ; "October"
                dq offset aNovember_1   ; "November"
                dq offset aDecember_1   ; "December"
aJanuary_1    db 'January',0        ; DATA XREF: sub_140001020+4
                ; .data:off_140011000
aFebruary_1   db 'February',0      ; DATA XREF: .data:0000000140011008
                align 4
aMarch_1     db 'March',0          ; DATA XREF: .data:0000000140011010
                align 4
aApril_1     db 'April',0          ; DATA XREF: .data:0000000140011018

```

Имена месяцев идут сразу после. Наша программа все-таки крошечная, так что здесь не так уж много данных (всего лишь названия месяцев) для расположения их в сегменте данных.

Но нужно заметить, что там может быть действительно *что угодно*, что линкер решит там расположить, случайным образом.

Так что будет если 12 будет передано в функцию? Вернется 13-й элемент таблицы. Посмотрим, как CPU обходится с байтами как с 64-битным значением:

Листинг 1.244: Исполняемый файл в IDA

```

off_140011000  dq offset qword_140011060
                ; DATA XREF: .text:0000000140001003
                dq offset aFebruary_1    ; "February"
                dq offset aMarch_1       ; "March"
                dq offset aApril_1      ; "April"
                dq offset aMay_1        ; "May"
                dq offset aJune_1       ; "June"

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

dq offset aJuly_1      ; "July"
dq offset aAugust_1   ; "August"
dq offset aSeptember_1 ; "September"
dq offset aOctober_1  ; "October"
dq offset aNovember_1 ; "November"
dq offset aDecember_1 ; "December"
qword_140011060 dq 797261756E614Ah ; DATA XREF: sub_140001020+4
; .data:off_140011000
aFebruary_1 db 'February',0 ; DATA XREF: .data:0000000140011008
align 4
aMarch_1 db 'March',0 ; DATA XREF: .data:0000000140011010

```

И это 0x797261756E614A. После этого, какая-то другая функция (вероятно, работающая со строками) попытается загружать байты по этому адресу, ожидая найти там Си-строку.

И скорее всего упадет, потому что это значение не выглядит как действительный адрес.

### Защита от переполнения массива

Если какая-нибудь неприятность может случиться, она случается

Закон Мерфи

Немного наивно ожидать что всякий программист, кто будет использовать вашу функцию или библиотеку, никогда не передаст аргумент больше 11.

Существует также хорошая философия «fail early and fail loudly» или «fail-fast», которая учит сообщать об ошибках как можно раньше и останавливаться.

Один из таких методов в Си/Си++ это макрос `assert()`.

Мы можем немного изменить нашу программу, чтобы она падала при передаче неверного значения:

Листинг 1.245: `assert()` добавлен

```

const char* get_month1_checked (int month)
{
    assert (month<12);
    return month1[month];
};

```

Макрос будет проверять на верные значения во время каждого старта функции и падать если выражение возвращает `false`.

Листинг 1.246: Оптимизирующий MSVC 2013 x64

```

$SG3143 DB 'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00H, '.', 00H
DB 'c', 00H, 00H, 00H
$SG3144 DB 'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00H, '<', 00H
DB '1', 00H, '2', 00H, 00H, 00H

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

month$ = 48
get_month1_checked PROC
$LN5:
    push    rbx
    sub     rsp, 32
    movsxd  rbx, ecx
    cmp     ebx, 12
    jl     SHORT $LN3@get_month1
    lea    rdx, OFFSET FLAT:$SG3143
    lea    rcx, OFFSET FLAT:$SG3144
    mov    r8d, 29
    call   _wassert
$LN3@get_month1:
    lea    rcx, OFFSET FLAT:month1
    mov    rax, QWORD PTR [rcx+rbx*8]
    add    rsp, 32
    pop    rbx
    ret    0
get_month1_checked ENDP

```

На самом деле, `assert()` это не функция, а макрос. Он проверяет условие и передает также номер строки и название файла в другую функцию, которая покажет эту информацию пользователю.

Мы видим, что здесь и имя файла и выражение закодировано в UTF-16.

Номер строки также передается (это 29).

Этот механизм, пожалуй, одинаковый во всех компиляторах.

Вот что делает GCC:

#### Листинг 1.247: Оптимизирующий GCC 4.9 x64

```

.LC1:
    .string "month.c"
.LC2:
    .string "month<12"

get_month1_checked:
    cmp     edi, 11
    jg     .L6
    movsx  rdi, edi
    mov    rax, QWORD PTR month1[0+rdi*8]
    ret

.L6:
    push   rax
    mov    ecx, OFFSET FLAT:__PRETTY_FUNCTION__.2423
    mov    edx, 29
    mov    esi, OFFSET FLAT:.LC1
    mov    edi, OFFSET FLAT:.LC2
    call   __assert_fail

__PRETTY_FUNCTION__.2423:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
.string "get_month1_checked"
```

Так что макрос в GCC также передает и имя функции, для удобства.

Ничего не бывает бесплатным и проверки на корректность тоже.

Это может замедлить работу вашей программы, особенно если макрос `assert()` используется в маленькой критичной ко времени функции.

Так что, например, MSVC оставляет проверки в отладочных сборках, но в окончательных сборках они исчезают.

Ядра Microsoft [Windows NT](#) также идут в видеборок «checked» и «free» <sup>135</sup>. В первых есть проверки на корректность аргументов (отсюда «checked»), а во вторых — нет (отсюда «free», т.е. «свободные» от проверок).

Разумеется, «checked»-ядро работает медленнее из-за всех этих проверок, поэтому его обычно используют только на время отладки драйверов, либо самого ядра.

### Доступ к определенному символу

К массиву указателей на строки можно обращаться так:

```
#include <stdio.h>

const char* month[]=
{
    "January", "February", "March", "April",
    "May", "June", "July", "August",
    "September", "October", "November", "December"
};

int main()
{
    // 4-й месяц, 5-й символ:
    printf ("%c\n", month[3][4]);
};
```

...так как, выражение `month[3]` имеет тип `const char*`. И затем, 5-й символ берется из этого выражения прибавлением 4-х байт к его адресу.

Кстати, список аргументов передаваемый в ф-цию `main()` имеет такой же тип:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf ("3-й аргумент, 2-й символ: %c\n", argv[3][1]);
};
```

<sup>135</sup> [msdn.microsoft.com/en-us/library/windows/hardware/ff543450\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/hardware/ff543450(v=vs.85).aspx)

Очень важно понимать, что не смотря на одинаковый синтаксис, всё это отличается от двумерных массивов, которые мы будем рассматривать позже.

Еще одна важная вещь, которую нужно отметить: адресуемые строки должны быть закодированы в системе, в которой каждый символ занимает один байт, как **ASCII!**<sup>136</sup> и расширенная **ASCII!**. UTF-8 здесь не будет работать.

### 1.26.6. Многомерные массивы

Внутри многомерный массив выглядит так же как и линейный.

Ведь память компьютера линейная, это одномерный массив. Но для удобства этот одномерный массив легко представить как многомерный.

К примеру, вот как элементы массива 3x4 расположены в одномерном массиве из 12 ячеек:

Смещение в памяти	элемент массива
0	[0][0]
1	[0][1]
2	[0][2]
3	[0][3]
4	[1][0]
5	[1][1]
6	[1][2]
7	[1][3]
8	[2][0]
9	[2][1]
10	[2][2]
11	[2][3]

Таблица 1.3: Двухмерный массив представляется в памяти как одномерный

Вот по каким адресам в памяти располагается каждая ячейка двумерного массива 3\*4:

0	1	2	3
4	5	6	7
8	9	10	11

Таблица 1.4: Адреса в памяти каждой ячейки двумерного массива

Чтобы вычислить адрес нужного элемента, сначала умножаем первый индекс (строку) на 4 (ширину массива), затем прибавляем второй индекс (столбец).

Это называется *row-major order*, и такой способ представления массивов и матриц используется по крайней мере в Си/Си++ и Python. Термин *row-major order*

<sup>136</sup>**ASCII!**



означает по-русски примерно следующее: «сначала записываем элементы первой строки, затем второй, ... и элементы последней строки в самом конце».

Другой способ представления называется *column-major order* (индексы массива используются в обратном порядке) и это используется по крайней мере в Фортране, MATLAB и R. Термин *column-major order* означает по-русски следующее: «сначала записываем элементы первого столбца, затем второго, ... и элементы последнего столбца в самом конце».

Какой из способов лучше? В терминах производительности и кэш-памяти, лучший метод организации данных это тот, при котором к данным обращаются последовательно.

Так что если ваша функция обращается к данным построчно, то *row-major order* лучше, и наоборот.

### Пример с двумерным массивом

Мы будем работать с массивом типа *char*. Это значит, что каждый элемент требует только одного байта в памяти.

### Пример с заполнением строки

Заполняем вторую строку значениями 0..3:

Листинг 1.248: Пример с заполнением строки

```
#include <stdio.h>

char a[3][4];

int main()
{
    int x, y;

    // очистить массив
    for (x=0; x<3; x++)
        for (y=0; y<4; y++)
            a[x][y]=0;

    // заполнить вторую строку значениями 0..3:
    for (y=0; y<4; y++)
        a[1][y]=y;
};
```

Все три строки обведены красным. Видно, что во второй теперь имеются байты 0, 1, 2 и 3:

Address	Hex dump
00C33370	00 00 00 00 00 01 02 03 00 00 00 00 00 00 00 00
00C33380	02 00 00 00 C3 66 47 4E C3 66 47 4E 00 00 00 00
00C33390	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C333A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C333B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Рис. 1.92: OllyDbg: массив заполнен

### Пример с заполнением столбца

Заполняем третий столбец значениями 0..2:

Листинг 1.249: Пример с заполнением столбца

```
#include <stdio.h>

char a[3][4];

int main()
{
    int x, y;

    // очистить массив
    for (x=0; x<3; x++)
        for (y=0; y<4; y++)
            a[x][y]=0;

    // заполнить третий столбец значениями 0..2:
    for (x=0; x<3; x++)
        a[x][2]=x;
};
```

Здесь также обведены красным три строки. Видно, что в каждой строке, на третьей позиции, теперь записаны 0, 1 и 2.

Address	Hex dump
01033380	00 00 00 00 00 00 01 00 00 00 02 00 02 00 00 00
01033390	00 00 00 00 1E 44 EF 31 1E 44 EF 31 00 00 00 00
010333A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
010333B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Рис. 1.93: OllyDbg: массив заполнен

### Работа с двумерным массивом как с одномерным

Мы можем легко убедиться, что можно работать с двумерным массивом как с одномерным, используя по крайней мере два метода:

```
#include <stdio.h>

char a[3][4];

char get_by_coordinates1 (char array[3][4], int a, int b)
{
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        return array[a][b];
};

char get_by_coordinates2 (char *array, int a, int b)
{
    // обращаться с входным массивом как с одномерным
    // 4 здесь это ширина массива
    return array[a*4+b];
};

char get_by_coordinates3 (char *array, int a, int b)
{
    // обращаться с входным массивом как с указателем,
    // вычислить адрес, получить значение оттуда
    // 4 здесь это ширина массива
    return *(array+a*4+b);
};

int main()
{
    a[2][3]=123;
    printf ("%d\n", get_by_coordinates1(a, 2, 3));
    printf ("%d\n", get_by_coordinates2(a, 2, 3));
    printf ("%d\n", get_by_coordinates3(a, 2, 3));
};

```

Компилируете<sup>137</sup> и запускаете: мы увидим корректные значения.

Очарователен результат работы MSVC 2013 — все три процедуры одинаковые!

#### Листинг 1.250: Оптимизирующий MSVC 2013 x64

```

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates3 PROC
; RCX=адрес массива
; RDX=a
; R8=b
        movsxd  rax, r8d
; EAX=b
        movsxd  r9, edx
; R9=a
        add     rax, rcx
; RAX=b+адрес массива
        movzx   eax, BYTE PTR [rax+r9*4]
; AL=загрузить байт по адресу RAX+R9*4=b+адрес массива+a*4=адрес
        массива+a*4+b
        ret     0
get_by_coordinates3 ENDP
array$ = 8

```

<sup>137</sup>Эта программа именно для Си а не Си++, компилируя в MSVC нужно сохранить файл с расширением .c

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

a$ = 16
b$ = 24
get_by_coordinates2 PROC
    movsxd  rax, r8d
    movsxd  r9, edx
    add     rax, rcx
    movzx   eax, BYTE PTR [rax+r9*4]
    ret     0
get_by_coordinates2 ENDP

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates1 PROC
    movsxd  rax, r8d
    movsxd  r9, edx
    add     rax, rcx
    movzx   eax, BYTE PTR [rax+r9*4]
    ret     0
get_by_coordinates1 ENDP

```

GCC сгенерировал практически одинаковые процедуры:

#### Листинг 1.251: Оптимизирующий GCC 4.9 x64

```

; RDI=адрес массива
; RSI=a
; RDX=b

get_by_coordinates1:
; расширить входные 32-битные значения "a" и "b" до 64-битных
    movsx   rsi, esi
    movsx   rdx, edx
    lea    rax, [rdi+rsi*4]
; RAX=RDI+RSI*4=адрес массива+a*4
    movzx   eax, BYTE PTR [rax+rdx]
; AL=загрузить байт по адресу RAX+RDX=адрес массива+a*4+b
    ret

get_by_coordinates2:
    lea    eax, [rdx+rsi*4]
; RAX=RDX+RSI*4=b+a*4
    cdqe
    movzx   eax, BYTE PTR [rdi+rax]
; AL=загрузить байт по адресу RDI+RAX=адрес массива+b+a*4
    ret

get_by_coordinates3:
    sal    esi, 2
; ESI=a<<2=a*4
; расширить входные 32-битные значения "a*4" и "b" до 64-битных
    movsx   rdx, edx
    movsx   rsi, esi
    add    rdi, rsi

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; RDI=RDI+RSI=адрес массива+a*4
  movzx  eax, BYTE PTR [rdi+rdx]
; AL=загрузить байт по адресу RDI+RDX=адрес массива+a*4+b
  ret

```

### Пример с трехмерным массивом

То же самое и для многомерных массивов. На этот раз будем работать с массивом типа *int*: каждый элемент требует 4 байта в памяти.

Попробуем:

Листинг 1.252: простой пример

```

#include <stdio.h>

int a[10][20][30];

void insert(int x, int y, int z, int value)
{
    a[x][y][z]=value;
};

```

### x86

В итоге (MSVC 2010):

Листинг 1.253: MSVC 2010

```

_DATA      SEGMENT
COMM      _a:DWORD:01770H
_DATA      ENDS
PUBLIC    _insert
_TEXT     SEGMENT
_x$ = 8           ; size = 4
_y$ = 12          ; size = 4
_z$ = 16          ; size = 4
_value$ = 20      ; size = 4
_insert   PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _x$[ebp]
    imul  eax, 2400           ; eax=600*4*x
    mov    ecx, DWORD PTR _y$[ebp]
    imul  ecx, 120           ; ecx=30*4*y
    lea   edx, DWORD PTR _a[eax+ecx] ; edx=a + 600*4*x + 30*4*y
    mov    eax, DWORD PTR _z$[ebp]
    mov    ecx, DWORD PTR _value$[ebp]
    mov    DWORD PTR [edx+eax*4], ecx ; *(edx+z*4)=значение
    pop   ebp
    ret   0
_insert   ENDP

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
_TEXT      ENDS
```

В принципе, ничего удивительного. В `insert()` для вычисления адреса нужного элемента массива три входных аргумента перемножаются по формуле  $address = 600 \cdot 4 \cdot x + 30 \cdot 4 \cdot y + 4z$ , чтобы представить массив трехмерным. Не забывайте также, что тип `int` 32-битный (4 байта), поэтому все коэффициенты нужно умножить на 4.

Листинг 1.254: GCC 4.4.1

```
public insert
insert  proc near

x      = dword ptr 8
y      = dword ptr 0Ch
z      = dword ptr 10h
value  = dword ptr 14h

        push    ebp
        mov     ebp, esp
        push    ebx
        mov     ebx, [ebp+x]
        mov     eax, [ebp+y]
        mov     ecx, [ebp+z]
        lea    edx, [eax+eax]      ; edx=y*2
        mov     eax, edx          ; eax=y*2
        shl    eax, 4             ; eax=(y*2)<<4 = y*2*16 = y*32
        sub    eax, edx          ; eax=y*32 - y*2=y*30
        imul   edx, ebx, 600      ; edx=x*600
        add    eax, edx          ; eax=eax+edx=y*30 + x*600
        lea    edx, [eax+ecx]     ; edx=y*30 + x*600 + z
        mov     eax, [ebp+value]
        mov     dword ptr ds:a[edx*4], eax ; *(a+edx*4)=значение
        pop     ebx
        pop     ebp
        retn
insert  endp
```

Компилятор GCC решил всё сделать немного иначе. Для вычисления одной из операций ( $30y$ ), GCC создал код, где нет самой операции умножения.

Происходит это так:  $(y + y) \ll 4 - (y + y) = (2y) \ll 4 - 2y = 2 \cdot 16 \cdot y - 2y = 32y - 2y = 30y$ . Таким образом, для вычисления  $30y$  используется только операция сложения, операция битового сдвига и операция вычитания. Это работает быстрее.

### ARM + Неоптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb)

Листинг 1.255: Неоптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb)

```
_insert
value  = -0x10
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

z      = -0xC
y      = -8
x      = -4

; выделить место в локальном стеке для 4 переменных типа int
SUB    SP, SP, #0x10
MOV    R9, 0xFC2 ; a
ADD    R9, PC
LDR.W  R9, [R9] ; получить указатель на массив
STR    R0, [SP,#0x10+x]
STR    R1, [SP,#0x10+y]
STR    R2, [SP,#0x10+z]
STR    R3, [SP,#0x10+value]
LDR    R0, [SP,#0x10+value]
LDR    R1, [SP,#0x10+z]
LDR    R2, [SP,#0x10+y]
LDR    R3, [SP,#0x10+x]
MOV    R12, 2400
MUL.W  R3, R3, R12
ADD    R3, R9
MOV    R9, 120
MUL.W  R2, R2, R9
ADD    R2, R3
LSLS   R1, R1, #2 ; R1=R1<<2
ADD    R1, R2
STR    R0, [R1] ; R1 - адрес элемента массива
; освободить блок в локальном стеке, выделенное для 4 переменных
ADD    SP, SP, #0x10
BX     LR

```

Неоптимизирующий LLVM сохраняет все переменные в локальном стеке, хотя это и избыточно.

Адрес элемента массива вычисляется по уже рассмотренной формуле.

### ARM + Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb)

Листинг 1.256: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb)

```

_insert
MOVW   R9, #0x10FC
MOV.W  R12, #2400
MOVT.W R9, #0
RSB.W  R1, R1, R1, LSL#4 ; R1 - y. R1=y<<4 - y = y*16 - y = y*15
ADD    R9, PC
LDR.W  R9, [R9] ; R9 = указатель на массив
MLA.W  R0, R0, R12, R9 ; R0 - x, R12 - 2400, R9 - указатель на a.
      R0=x*2400 + указатель на a
ADD.W  R0, R0, R1, LSL#3 ; R0 = R0+R1<<3 = R0+R1*8 = x*2400 + указатель на a |
      + y*15*8 =
      ; указатель на a + y*30*4 + x*600*4
STR.W  R3, [R0,R2,LSL#2] ; R2 - z, R3 - значение. адрес=R0+z*4 =
      ; указатель на a + y*30*4 + x*600*4 + z*4

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

BX LR

Тут используются уже описанные трюки для замены умножения на операции сдвига, сложения и вычитания.

Также мы видим новую для себя инструкцию RSB (*Reverse Subtract*). Она работает так же, как и SUB, только меняет операнды местами.

Зачем? SUB и RSB это те инструкции, ко второму операнду которых можно применить коэффициент сдвига, как мы видим и здесь: (LSL#4). Но этот коэффициент можно применить только ко второму операнду.

Для коммутативных операций, таких как сложение или умножение, операнды можно менять местами и это не влияет на результат.

Но вычитание — операция некоммутативная, так что для этих случаев существует инструкция RSB.

## MIPS

Мой пример такой крошечный, что компилятор GCC решил разместить массив *a* в 64KiB-области, адресуемой при помощи Global Pointer.

Листинг 1.257: Оптимизирующий GCC 4.4.5 (IDA)

```
insert:
; $a0=x
; $a1=y
; $a2=z
; $a3=значение
                sll    $v0, $a0, 5
; $v0 = $a0<<5 = x*32
                sll    $a0, 3
; $a0 = $a0<<3 = x*8
                addu   $a0, $v0
; $a0 = $a0+$v0 = x*8+x*32 = x*40
                sll    $v1, $a1, 5
; $v1 = $a1<<5 = y*32
                sll    $v0, $a0, 4
; $v0 = $a0<<4 = x*40*16 = x*640
                sll    $a1, 1
; $a1 = $a1<<1 = y*2
                subu   $a1, $v1, $a1
; $a1 = $v1-$a1 = y*32-y*2 = y*30
                subu   $a0, $v0, $a0
; $a0 = $v0-$a0 = x*640-x*40 = x*600
                la     $gp, __gnu_local_gp
                addu   $a0, $a1, $a0
; $a0 = $a1+$a0 = y*30+x*600
                addu   $a0, $a2
; $a0 = $a0+$a2 = y*30+x*600+z
; загрузить адрес таблицы:
                lw     $v0, (a & 0xFFFF)($gp)
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

; умножить индекс на 4 для поиска элемента таблицы:
    sll    $a0, 2
; сложить умноженный индекс и адрес таблицы:
    addu   $a0, $v0, $a0
; записать значение в таблицу и вернуть управление:
    jr     $ra
    sw     $a3, 0($a0)

.comm a:0x1770

```

### Узнать размеры многомерного массива

Если в ф-цию обработки строки передать массив символов, внутри самой ф-ции невозможно определить размер входного массива. Точно также, в ф-ции, обрабатывающую двухмерный массив, только один размер может быть определен.

Например:

```

int get_element(int array[10][20], int x, int y)
{
    return array[x][y];
};

int main()
{
    int array[10][20];

    get_element(array, 4, 5);
};

```

... если это скомпилировать (любым компилятором) и затем декомпилировать в Hex-Rays:

```

int get_element(int *array, int x, int y)
{
    return array[20 * x + y];
}

```

Нет никакого способа узнать размер первого измерения. Если переданное значение  $x$  слишком большое, произойдет переполнение буфера, и прочтается элемент из какого-то случайного места в памяти.

И трехмерный массив:

```

int get_element(int array[10][20][30], int x, int y, int z)
{
    return array[x][y][z];
};

int main()
{
    int array[10][20][30];
}

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    get_element(array, 4, 5, 6);
};

```

Hex-Rays:

```

int get_element(int *array, int x, int y, int z)
{
    return array[600 * x + z + 30 * y];
}

```

И снова, можно узнать размеры только двух измерений из трех.

### Ещё примеры

Компьютерный экран представляет собой двумерный массив, но видеобuffer это линейный одномерный массив. Мы рассматриваем это здесь: ?? (стр. ??).

Еще один пример в этой книге это игра “Сапер”: её поле это тоже двумерный массив: ?? (стр. ??).

### 1.26.7. Набор строк как двумерный массив

Снова вернемся к примеру, который возвращает название месяца: листинг.??  
Как видно, нужна как минимум одна операция загрузки из памяти для подготовки указателя на строку, состоящую из имени месяца.

Возможно ли избавиться от операции загрузки из памяти?

Да, если представить список строк как двумерный массив:

```

#include <stdio.h>
#include <assert.h>

const char month2[12][10]=
{
    { 'J','a','n','u','a','r','y', 0, 0, 0 },
    { 'F','e','b','r','u','a','r','y', 0, 0 },
    { 'M','a','r','c','h', 0, 0, 0, 0, 0 },
    { 'A','p','r','i','l', 0, 0, 0, 0, 0 },
    { 'M','a','y', 0, 0, 0, 0, 0, 0 },
    { 'J','u','n','e', 0, 0, 0, 0, 0, 0 },
    { 'J','u','l','y', 0, 0, 0, 0, 0, 0 },
    { 'A','u','g','u','s','t', 0, 0, 0, 0 },
    { 'S','e','p','t','e','m','b','e','r', 0 },
    { 'O','c','t','o','b','e','r', 0, 0, 0 },
    { 'N','o','v','e','m','b','e','r', 0, 0 },
    { 'D','e','c','e','m','b','e','r', 0, 0 }
};

// в пределах 0..11
const char* get_month2 (int month)
{

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
    return &month2[month][0];
};
```

Вот что получаем:

Листинг 1.258: Оптимизирующий MSVC 2013 x64

```
month2 DB    04aH
        DB    061H
        DB    06eH
        DB    075H
        DB    061H
        DB    072H
        DB    079H
        DB    00H
        DB    00H
        DB    00H
...
get_month2 PROC
; расширить входное значение до 64-битного, учитывая знак
    movsxd    rax, ecx
    lea      rcx, QWORD PTR [rax+rax*4]
; RCX=месяц*4=месяц*5
    lea      rax, OFFSET FLAT:month2
; RAX=указатель на таблицу
    lea      rax, QWORD PTR [rax+rcx*2]
; RAX=указатель на таблицу + RCX*2=указатель на таблицу + месяц*5*2=указатель
  на таблицу + месяц*10
    ret      0
get_month2 ENDP
```

Здесь нет обращений к памяти вообще. Эта функция только вычисляет место, где находится первый символ названия месяца:

*pointer\_to\_the\_table+month\*10*. Там также две инструкции LEA, которые работают как несколько инструкций MUL и MOV.

Ширина массива — 10 байт. Действительно, самая длинная строка это «September» (9 байт) плюс оконечивающий ноль, получается 10 байт.

Остальные названия месяцев дополнены нулевыми байтами, чтобы они занимали столько же места (10 байт).

Таким образом, наша функция и работает быстрее, потому что все строки начинаются с тех адресов, которые легко вычислить.

Оптимизирующий GCC 4.9 может ещё короче:

Листинг 1.259: Оптимизирующий GCC 4.9 x64

```
movsx    rdi, edi
lea      rax, [rdi+rdi*4]
lea      rax, month2[rax+rax]
ret
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

LEA здесь также используется для умножения на 10.

Неоптимизирующие компиляторы делают умножение по-разному.

Листинг 1.260: Неоптимизирующий GCC 4.9 x64

```

get_month2:
    push    rbp
    mov     rbp, rsp
    mov     DWORD PTR [rbp-4], edi
    mov     eax, DWORD PTR [rbp-4]
    movsx   rdx, eax
; RDX = входное значение, расширенное учитывая знак
    mov     rax, rdx
; RAX = месяц
    sal    rax, 2
; RAX = месяц<<2 = месяц*4
    add    rax, rdx
; RAX = RAX+RDX = месяц*4+месяц = месяц*5
    add    rax, rax
; RAX = RAX*2 = месяц*5*2 = месяц*10
    add    rax, OFFSET FLAT:month2
; RAX = месяц*10 + указатель на таблицу
    pop    rbp
    ret

```

Неоптимизирующий MSVC просто использует инструкцию IMUL:

Листинг 1.261: Неоптимизирующий MSVC 2013 x64

```

month$ = 8
get_month2 PROC
    mov     DWORD PTR [rsp+8], ecx
    movsxd  rax, DWORD PTR month$[rsp]
; RAX = расширенное до 64-битного входное значение, учитывая знак
    imul   rax, rax, 10
; RAX = RAX*10
    lea    rcx, OFFSET FLAT:month2
; RCX = указатель на таблицу
    add    rcx, rax
; RCX = RCX+RAX = указатель на таблицу+month*10
    mov     rax, rcx
; RAX = указатель на таблицу+месяц*10
    mov     ecx, 1
; RCX = 1
    imul   rcx, rcx, 0
; RCX = 1*0 = 0
    add    rax, rcx
; RAX = указатель на таблицу+месяц*10 + 0 = указатель на таблицу+месяц*10
    ret    0
get_month2 ENDP

```

Но вот что странно: зачем добавлять умножение на ноль и добавлять ноль к конечному результату?

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Это выглядит как странность кодегенератора компилятора, который не был покрыт тестами компилятора. Но так или иначе, итоговый код работает корректно. Мы сознательно рассматриваем такие фрагменты кода, чтобы читатель понимал, что иногда не нужно ломать себе голову над подобными артефактами компиляторов.

### 32-bit ARM

Оптимизирующий Keil для режима Thumb использует инструкцию умножения MULS:

Листинг 1.262: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
; R0 = месяц
    MOVS    r1,#0xa
; R1 = 10
    MULS   r0,r1,r0
; R0 = R1*R0 = 10*месяц
    LDR    r1,|L0.68|
; R1 = указатель на таблицу
    ADDS   r0,r0,r1
; R0 = R0+R1 = 10*месяц + указатель на таблицу
    BX     lr
```

Оптимизирующий Keil для режима ARM использует операции сложения и сдвига:

Листинг 1.263: Оптимизирующий Keil 6/2013 (Режим ARM)

```
; R0 = месяц
    LDR    r1,|L0.104|
; R1 = указатель на таблицу
    ADD    r0,r0,r0,LSL #2
; R0 = R0+R0<<2 = R0+R0*4 = месяц*5
    ADD    r0,r1,r0,LSL #1
; R0 = R1+R0<<2 = указатель на таблицу + месяц*5*2 = указатель на таблицу +
    месяц*10
    BX     lr
```

### ARM64

Листинг 1.264: Оптимизирующий GCC 4.9 ARM64

```
; W0 = месяц
    sxtw   x0, w0
; X0 = расширить входное значение учитывая знак
    adrp   x1, .LANCHOR1
    add    x1, x1, :lo12:.LANCHOR1
; X1 = указатель на таблицу
    add    x0, x0, x0, lsl 2
; X0 = X0+X0<<2 = X0+X0*4 = X0*5
    add    x0, x1, x0, lsl 1
; X0 = X1+X0<<1 = X1+X0*2 = указатель на таблицу + X0*10
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

ret

SXTW используется для знакового расширения и расширения входного 32-битного значения в 64-битное и сохранения его в X0.

Пара ADRP/ADD используется для загрузки адреса таблицы.

У инструкции ADD также есть суффикс LSL, что помогает с умножением.

## MIPS

Листинг 1.265: Оптимизирующий GCC 4.4.5 (IDA)

```

        .globl get_month2
get_month2:
; $a0=месяц
        sll    $v0, $a0, 3
; $v0 = $a0<<3 = месяц*8
        sll    $a0, 1
; $a0 = $a0<<1 = месяц*2
        addu   $a0, $v0
; $a0 = месяц*2+месяц*8 = месяц*10
; загрузить адрес таблицы:
        la     $v0, month2
; сложить адрес таблицы и вычисленный индекс и вернуть управление:
        jr     $ra
        addu   $v0, $a0

month2:
        .ascii "January"<0>
        .byte 0, 0
aFebruary:
        .ascii "February"<0>
        .byte 0
aMarch:
        .ascii "March"<0>
        .byte 0, 0, 0, 0
aApril:
        .ascii "April"<0>
        .byte 0, 0, 0, 0
aMay:
        .ascii "May"<0>
        .byte 0, 0, 0, 0, 0, 0
aJune:
        .ascii "June"<0>
        .byte 0, 0, 0, 0, 0
aJuly:
        .ascii "July"<0>
        .byte 0, 0, 0, 0, 0
aAugust:
        .ascii "August"<0>
        .byte 0, 0, 0
aSeptember:
        .ascii "September"<0>
aOctober:
        .ascii "October"<0>
        .byte 0, 0
aNovember:
        .ascii "November"<0>
        .byte 0
aDecember:
        .ascii "December"<0>
        .byte 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

## Вывод

Это немного олд-скульная техника для хранения текстовых строк. Много такого можно найти в Oracle RDBMS, например. Трудно сказать, стоит ли оно того на современных компьютерах. Так или иначе, это был хороший пример массивов, поэтому он был добавлен в эту книгу.

### 1.26.8. Вывод

Массив это просто набор значений в памяти, расположенных рядом друг с другом.

Это справедливо для любых типов элементов, включая структуры.

Доступ к определенному элементу массива это просто вычисление его адреса.

Итак, указатель на массив и адрес первого элемента — это одно и то же. Вот почему выражения `ptr[0]` и `*ptr` в Си/Си++ равноценны. Любопытно что Hex-Rays часто заменяет первое вторым. Он делает это в тех случаях, когда не знает, что имеет дело с указателем на целый массив, и думает, что это указатель только на одну переменную.

### 1.26.9. Упражнения

- <http://challenges.re/62>
- <http://challenges.re/63>
- <http://challenges.re/64>
- <http://challenges.re/65>
- <http://challenges.re/66>

## 1.27. Пример: ошибка в Angband

В сравнительно древней rogue-like игре 90-х <sup>138</sup> была ошибка в духе “Пикника на обочине” Стругацких или сериала “Потерянная комната”<sup>139</sup>:

Версия frog-knows изобиловала ошибками. Самая смешная из них повлекла хитрую технику обмана игры, которую назвали “mushroom farming” – разведение грибов. Если в лабиринте оказывалось больше определенного числа (около пятисот) предметов, игра ломалась, и старые предметы помногу превращались в предметы, брошенные на пол. Соответственно, игрок шел в лабиринт, делал там такие продольные канавки (специальным заклинанием), и ходил вдоль канавок, создавая грибы еще другим

<sup>138</sup>[https://en.wikipedia.org/wiki/Angband\\_\(video\\_game\)](https://en.wikipedia.org/wiki/Angband_(video_game)), <http://rephial.org/>

<sup>139</sup><http://archive.is/oIQyL>

специальным заклинанием. Когда грибов было достаточно много, гражданин клал и брал, клал и брал какой-нибудь полезный предмет, и грибы один за другим превращались в этот предмет. После этого игрок возвращался с сотнями копий полезного предмета.

( Миша "tiphareth" Вербицкий, <http://imperium.lenin.ru/СЕВЕР/arc/3/lightmusic/light.htm> )

Еще из usenet-a:

From: be...@uswest.com (George Bell)  
Subject: [Angband] Multiple artifact copies found (bug?)  
Date: Fri, 23 Jul 1993 15:55:08 GMT

Up to 2000 ft I found only 4 artifacts, now my house is littered with the suckers (FYI, most I've gotten from killing nasties, like Dracoliches and ↵ the like). Something really weird is happening now, as I found multiple copies of the same artifact! My half-elf ranger is down at 2400 ft on one level which is particularly nasty. There is a graveyard plus monsters surrounded by permanent rock and 2 or 3 other special monster rooms! I did so much slashing with my favorite weapon, Crisdurian, that I filled several rooms nearly to the brim with treasure (as usual, mostly junk).

Then, when I found a way into the big vault, I noticed some of the treasure had already been identified (in fact it looked strangely familiar!). Then ↵ I found \*two\* Short Swords named Sting (1d6) (+7,+8), and I just ran across a third copy! I have seen multiple copies of Gurthang on this level as well. Is there some limit on the number of items per level which I have exceeded? This sounds reasonable as all multiple copies I have seen come from this ↵ level.

I'm playing PC angband. Anybody else had this problem?

-George Bell

Help! I need a Rod of Restore Life Levels, if there is such a thing. ↵  
↳ These Graveyards are nasty (Black Reavers and some speed 2 wraith in particular).

( <https://groups.google.com/forum/#!original/rec.games.moria/jItmfrdGyL8/8csctQqA7PQJ> )

From: Ceri <cm...@andrew.cmu.edu>  
Subject: Re: [Angband] Multiple artifact copies found (bug?)  
Date: Fri, 23 Jul 1993 23:32:20 -0400

welcome to the mush bug. if there are more than 256 items on the floor, things start duplicating. learn to harness this power and you will win shortly :>

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



--Rick

( [google groups](#) )

From: nwe...@soda.berkeley.edu (Nicholas C. Weaver)  
 Subject: Re: [Angband] Multiple artifact copies found (bug?)  
 Date: 24 Jul 1993 18:18:05 GMT

In article <74348474...@unix1.andrew.cmu.edu> Ceri <cm...@andrew.cmu.edu> ↵  
 ↳ writes:

>welcome to the mush bug. if there are more than 256 items  
 >on the floor, things start duplicating. learn to harness  
 >this power and you will win shortly :>  
 >  
 >--Rick

QuesTion on this. Is it only the first 256 items which get  
 duplicated? What about the original items? Etc ETC ETC...

Oh, for those who like to know about bugs, though, the -n option  
 (start new character) has the following behavior:

(this is in version 2.4.Frog.knows on unix)

If you hit controll-p, you keep your old stats.

Y0u loose all record of artifacts founds and named monsters killed.

Y0u loose all items you are carrying (they get turned into error in  
 objid(s) ).

You loose your gold.

You KEEP all the stuff in your house.

If you kill something, and then quaff a potion of restore life  
 levels, you are back up to where you were before in EXPERIENCE POINTS!!

Gaining spells will not work right after this, unless you have a  
 gain int item (for spellcasters) or gain wis item (for priests/palidans), ↵  
 ↳ in

which case after performing the above, then take the item back on and off,  
 you will be able to learn spells normally again.

This can be exploited, if you are a REAL H0ZER (like me), into  
 getting multiple artifacts early on. Just get to a level where you can  
 pound wormtongue into the ground, kill him, go up, drop your stuff in your  
 house, buy a few potions of restore exp and high value spellbooks with your  
 leftover gold, angband -n yourself back to what you were before, and repeat  
 the process. Yes, you CAN kill wormtongue multiple times. :)

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения,  
 пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    This also allows the creation of a human rogue with dunedain ↗
    ↘ warrior
starting stats.

    Of course, such practices are evil, vile, and disgusting. I take ↗
    ↘ no
liability for the results of spreading this information. Yeah, it's ↗
    ↘ another
bug to go onto the pile.
--
Nicholas C. Weaver      perpetual ensign guppy      nwe...@soda.berkeley.↗
    ↘ edu
It is a tale, told by an idiot, full of sound and fury, .signifying ↗
    ↘ nothing.
    Since C evolved out of B, and a C+ is close to a B,
    does that mean that C++ is a devolution of the language?

```

( <https://groups.google.com/forum/#!original/rec.games.moria/jItmfrdGyL8/FoQeiccewHAJ> )

Весь тред.

Автор этих строк нашел версию с ошибкой (2.4 fk) <sup>140</sup>, и мы легко можем увидеть, как определены глобальные массивы:

```

/* Number of dungeon objects */
#define MAX_DUNGEON_OBJ  423

...

int16 sorted_objects[MAX_DUNGEON_OBJ];

/* Identified objects flags */
int8u object_ident[OBJECT_IDENT_SIZE];
int16 t_level[MAX_OBJ_LEVEL+1];
inven_type t_list[MAX_TALLOC];
inven_type inventory[INVEN_ARRAY_SIZE];

```

Видимо, это и есть причина. Константа MAX\_DUNGEON\_OBJ слишком маленькая. Наверное, авторам следовало бы использовать связные списки, или иные структуры данных, без ограничений на размер. Но с массивами работать проще.

Еще один пример переполнения буфера в глобальных массивах: ?? (стр. ??).

## 1.28. Работа с отдельными битами

Немало функций задают различные флаги в аргументах при помощи битовых полей<sup>141</sup>.

<sup>140</sup><http://rephial.org/release/2.4.fk>, <https://yurichev.com/mirrors/angband-2.4.fk.tar>

<sup>141</sup>bit fields в англоязычной литературе

Наверное, вместо этого можно было бы использовать набор переменных типа *bool*, но это было бы не очень экономно.

### 1.28.1. Проверка какого-либо бита

#### x86

Например в Win32 API:

```
HANDLE fh;

fh=CreateFile ("file", GENERIC_WRITE | GENERIC_READ, ↵
↵ FILE_SHARE_READ, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
```

Получаем (MSVC 2010):

Листинг 1.266: MSVC 2010

```
push    0
push    128          ; 00000080H
push    4
push    0
push    1
push    -1073741824 ; c0000000H
push    OFFSET $SG78813
call    DWORD PTR __imp__CreateFileA@28
mov     DWORD PTR _fh$[ebp], eax
```

Заглянем в файл WinNT.h:

Листинг 1.267: WinNT.h

```
#define GENERIC_READ          (0x80000000L)
#define GENERIC_WRITE         (0x40000000L)
#define GENERIC_EXECUTE      (0x20000000L)
#define GENERIC_ALL           (0x10000000L)
```

Всё ясно,  $\text{GENERIC\_READ} | \text{GENERIC\_WRITE} = 0x80000000 | 0x40000000 = 0xC0000000$ , и это значение используется как второй аргумент для функции `CreateFile()` <sup>142</sup>.

Как `CreateFile()` будет проверять флаги? Заглянем в `KERNEL32.DLL` от Windows XP SP3 x86 и найдем в функции `CreateFileW()` в том числе и такой фрагмент кода:

Листинг 1.268: KERNEL32.DLL (Windows XP SP3 x86)

```
.text:7C83D429    test    byte ptr [ebp+dwDesiredAccess+3], 40h
.text:7C83D42D    mov     [ebp+var_8], 1
.text:7C83D434    jz     short loc_7C83D417
.text:7C83D436    jmp    loc_7C810817
```

<sup>142</sup>[msdn.microsoft.com/en-us/library/aa363858\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa363858(VS.85).aspx)

Здесь мы видим инструкцию TEST. Впрочем, она берет не весь второй аргумент функции, а только его самый старший байт (`ebp+dwDesiredAccess+3`) и проверяет его на флаг `0x40` (имеется ввиду флаг `GENERIC_WRITE`).

TEST это то же что и AND, только без сохранения результата (вспомните что CMP это то же что и SUB, только без сохранения результатов (?? (стр. ??))).

Логика данного фрагмента кода примерно такая:

```
if ((dwDesiredAccess&0x40000000) == 0) goto loc_7C83D417
```

Если после операции AND останется этот бит, то флаг ZF не будет поднят и условный переход JZ не сработает. Переход возможен, только если в переменной `dwDesiredAccess` отсутствует бит `0x40000000` — тогда результат AND будет 0, флаг ZF будет поднят и переход сработает.

Попробуем GCC 4.4.1 и Linux:

```
#include <stdio.h>
#include <fcntl.h>

void main()
{
    int handle;

    handle=open ("file", O_RDWR | O_CREAT);
};
```

Получим:

Листинг 1.269: GCC 4.4.1

```
main                public main
                   proc near
var_20              = dword ptr -20h
var_1C              = dword ptr -1Ch
var_4               = dword ptr -4

                   push    ebp
                   mov     ebp, esp
                   and     esp, 0FFFFFFF0h
                   sub     esp, 20h
                   mov     [esp+20h+var_1C], 42h
                   mov     [esp+20h+var_20], offset aFile ; "file"
                   call    _open
                   mov     [esp+20h+var_4], eax
                   leave
                   retn
main                endp
```

Заглянем в реализацию функции `open()` в библиотеке `libc.so.6`, но обнаружим что там только системный вызов:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Листинг 1.270: open() (libc.so.6)

```
.text:000BE69B    mov     edx, [esp+4+mode] ; mode
.text:000BE69F    mov     ecx, [esp+4+flags] ; flags
.text:000BE6A3    mov     ebx, [esp+4+filename] ; filename
.text:000BE6A7    mov     eax, 5
.text:000BE6AC    int     80h                ; LINUX - sys_open
```

Значит, битовые поля флагов open() проверяются где-то в ядре Linux.

Разумеется, и стандартные библиотеки Linux и ядро Linux можно получить в виде исходников, но нам интересно попробовать разобраться без них.

При системном вызове sys\_open управление в конечном итоге передается в do\_sys\_open в ядре Linux 2.6. Оттуда — в do\_filp\_open() (эта функция находится в исходниках ядра в файле fs/namei.c).

N.B. Помимо передачи параметров функции через стек, существует также возможность передавать некоторые из них через регистры. Такое соглашение о вызове называется fastcall (?? (стр. ??)). Оно работает немного быстрее, так как для чтения аргументов процессору не нужно обращаться к стеку, лежащему в памяти. В GCC есть опция *regparm*<sup>143</sup>, и с её помощью можно задать, сколько аргументов можно передать через регистры.

Ядро Linux 2.6 собирается с опцией *-mregparm=3*<sup>144 145</sup>.

Для нас это означает, что первые три аргумента функции будут передаваться через регистры EAX, EDX и ECX, а остальные через стек. Разумеется, если аргументов у функции меньше трех, то будет задействована только часть этих регистров.

Итак, качаем ядро 2.6.31, собираем его в Ubuntu, открываем в **IDA!**, находим функцию do\_filp\_open(). В начале мы увидим что-то такое (комментарии мои):

Листинг 1.271: do\_filp\_open() (linux kernel 2.6.31)

```
do_filp_open    proc near
...
                push    ebp
                mov     ebp, esp
                push    edi
                push    esi
                push    ebx
                mov     ebx, ecx
                add     ebx, 1
                sub     esp, 98h
                mov     esi, [ebp+arg_4] ; acc_mode (пятый аргумент)
                test    bl, 3
                mov     [ebp+var_80], eax ; dfd (первый аргумент)
                mov     [ebp+var_7C], edx ; pathname (второй аргумент)
                mov     [ebp+var_78], ecx ; open_flag (третий аргумент)
                jnz     short loc_C01EF684
```

<sup>143</sup>[ohse.de/uwe/articles/gcc-attributes.html#func-regparm](http://ohse.de/uwe/articles/gcc-attributes.html#func-regparm)

<sup>144</sup>[kernelnewbies.org/Linux\\_2\\_6\\_20#head-042c62f290834eb1fe0a1942bbf5bb9a4accbc8f](http://kernelnewbies.org/Linux_2_6_20#head-042c62f290834eb1fe0a1942bbf5bb9a4accbc8f)

<sup>145</sup>См. также файл arch/x86/include/asm/calling.h в исходниках ядра

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
mov    ebx, ecx    ; ebx <- open_flag
```

GCC сохраняет значения первых трех аргументов в локальном стеке. Иначе, если эти три регистра не трогать вообще, то функции компилятора, распределяющей переменные по регистрам (так называемый [register allocator](#)), будет очень тесно.

Далее находим примерно такой фрагмент кода:

Листинг 1.272: do\_filp\_open() (linux kernel 2.6.31)

```
loc_C01EF684:    ; CODE XREF: do_filp_open+4F
                test    bl, 40h    ; 0_CREAT
                jnz    loc_C01EF810
                mov    edi, ebx
                shr    edi, 11h
                xor    edi, 1
                and    edi, 1
                test   ebx, 10000h
                jz     short loc_C01EF6D3
                or     edi, 2
```

0x40 — это значение макроса 0\_CREAT. open\_flag проверяется на наличие бита 0x40 и если бит равен 1, то выполняется следующая за JNZ инструкция.

## ARM

В ядре Linux 3.8.0 бит 0\_CREAT проверяется немного иначе.

Листинг 1.273: linux kernel 3.8.0

```
struct file *do_filp_open(int dfd, struct filename *pathname,
                          const struct open_flags *op)
{
    ...
    filp = path_openat(dfd, pathname, &nd, op, flags | LOOKUP_RCU);
    ...
}

static struct file *path_openat(int dfd, struct filename *pathname,
                                struct nameidata *nd, const struct open_flags *op, int ↵
                                ↵ flags)
{
    ...
    error = do_last(nd, &path, file, op, &opened, pathname);
    ...
}

static int do_last(struct nameidata *nd, struct path *path,
                  struct file *file, const struct open_flags *op,
                  int *opened, struct filename *name)
{
    ...
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    if (!(open_flag & O_CREAT)) {
    ...
        error = lookup_fast(nd, path, &inode);
    ...
    } else {
    ...
        error = complete_walk(nd);
    }
    ...
}

```

Вот как это выглядит в **IDA!**, ядро скомпилированное для режима ARM:

Листинг 1.274: do\_last() из vmlinux (IDA)

```

...
.text:C0169EA8    MOV     R9, R3 ; R3 - (4th argument) open_flag
...
.text:C0169ED4    LDR     R6, [R9] ; R6 - open_flag
...
.text:C0169F68    TST     R6, #0x40 ; jumptable C0169F00 default case
.text:C0169F6C    BNE     loc_C016A128
.text:C0169F70    LDR     R2, [R4,#0x10]
.text:C0169F74    ADD     R12, R4, #8
.text:C0169F78    LDR     R3, [R4,#0xC]
.text:C0169F7C    MOV     R0, R4
.text:C0169F80    STR     R12, [R11,#var_50]
.text:C0169F84    LDRB    R3, [R2,R3]
.text:C0169F88    MOV     R2, R8
.text:C0169F8C    CMP     R3, #0
.text:C0169F90    ORRNE   R1, R1, #3
.text:C0169F94    STRNE   R1, [R4,#0x24]
.text:C0169F98    ANDS    R3, R6, #0x200000
.text:C0169F9C    MOV     R1, R12
.text:C0169FA0    LDRNE   R3, [R4,#0x24]
.text:C0169FA4    ANDNE   R3, R3, #1
.text:C0169FA8    EORNE   R3, R3, #1
.text:C0169FAC    STR     R3, [R11,#var_54]
.text:C0169FB0    SUB     R3, R11, #-var_38
.text:C0169FB4    BL      lookup_fast
...
.text:C016A128    loc_C016A128 ; CODE XREF: do_last.isra.14+DC
.text:C016A128    MOV     R0, R4
.text:C016A12C    BL      complete_walk
...

```

TST это аналог инструкции TEST в x86. Мы можем «узнать» визуально этот фрагмент кода по тому что в одном случае исполнится функция `lookup_fast()`, а в другом `complete_walk()`. Это соответствует исходному коду функции `do_last()`. Макрос `O_CREAT` здесь так же равен `0x40`.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

## 1.28.2. Установка и сброс отдельного бита

Например:

```
#include <stdio.h>

#define IS_SET(flag, bit)      ((flag) & (bit))
#define SET_BIT(var, bit)     ((var) |= (bit))
#define REMOVE_BIT(var, bit)  ((var) &= ~(bit))

int f(int a)
{
    int rt=a;

    SET_BIT (rt, 0x4000);
    REMOVE_BIT (rt, 0x200);

    return rt;
};

int main()
{
    f(0x12340678);
};
```

### x86

#### Неоптимизирующий MSVC

Имеем в итоге (MSVC 2010):

Листинг 1.275: MSVC 2010

```
_rt$ = -4          ; size = 4
_a$ = 8           ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _a$[ebp]
    mov     DWORD PTR _rt$[ebp], eax
    mov     ecx, DWORD PTR _rt$[ebp]
    or      ecx, 16384          ; 00004000H
    mov     DWORD PTR _rt$[ebp], ecx
    mov     edx, DWORD PTR _rt$[ebp]
    and     edx, -513          ; ffffffffH
    mov     DWORD PTR _rt$[ebp], edx
    mov     eax, DWORD PTR _rt$[ebp]
    mov     esp, ebp
    pop     ebp
    ret     0
_f ENDP
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



---

Инструкция OR здесь устанавливает в регистре один бит, игнорируя остальные биты-единицы.

А AND сбрасывает некий бит. Можно также сказать, что AND здесь копирует все биты, кроме одного. Действительно, во втором операнде AND выставлены в единицу те биты, которые нужно сохранить, кроме одного, копировать который мы не хотим (и который 0 в битовой маске). Так проще понять и запомнить.

**OllyDbg**

Попробуем этот пример в OllyDbg. Сначала, посмотрим на двоичное представление используемых нами констант:

0x200 (0b000000000000000000000000000000001000000000) (т.е. 10-й бит (считая с первого)).

Инвертированное 0x200 это 0xFFFFDFF  
(0b1111111111111111111111110111111111).

0x4000 (0b000000000000000000000000000000001000000000000000) (т.е. 15-й бит).

Входное значение это: 0x12340678

(0b100100011010000000110011111000). Видим, как оно загрузилось:

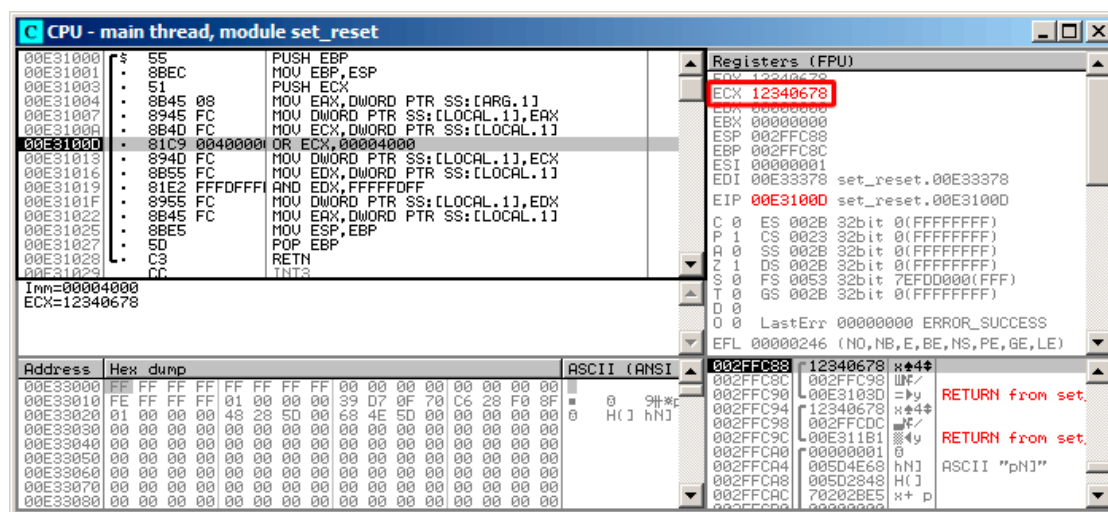


Рис. 1.94: OllyDbg: значение загружено в ECX

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

OR исполнилась:

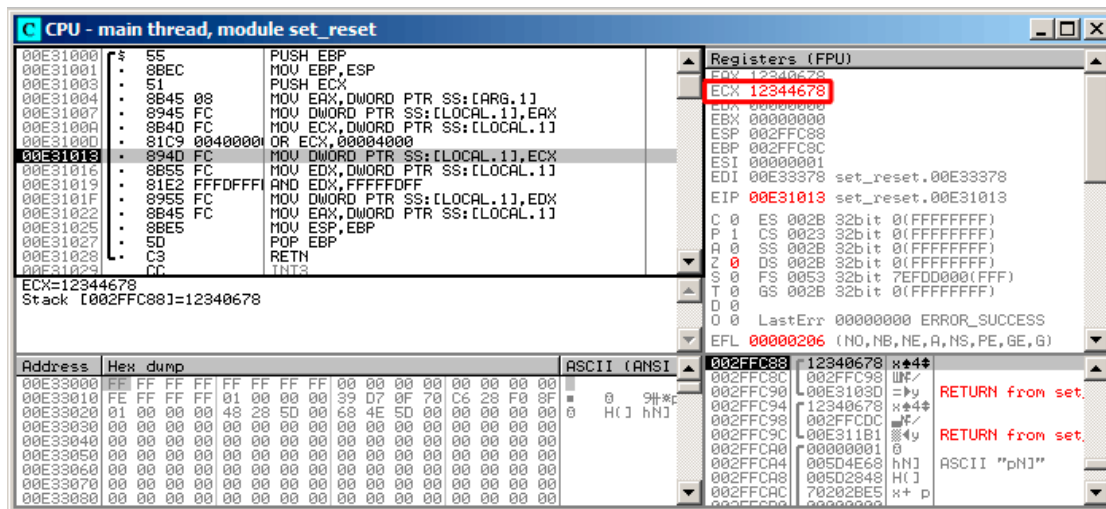


Рис. 1.95: OllyDbg: OR сработал

15-й бит выставлен: 0x12344678  
(0b10010001101000100011001111000).

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Значение перезагружается снова (потому что использовался режим компилятора без оптимизации):

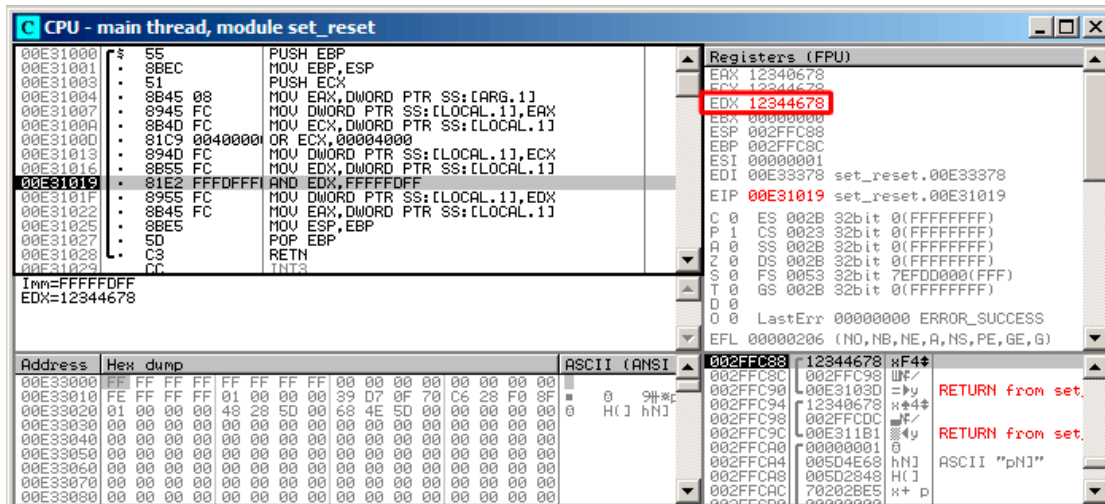


Рис. 1.96: OllyDbg: значение перезагрузилось в EDX

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

AND исполнилась:

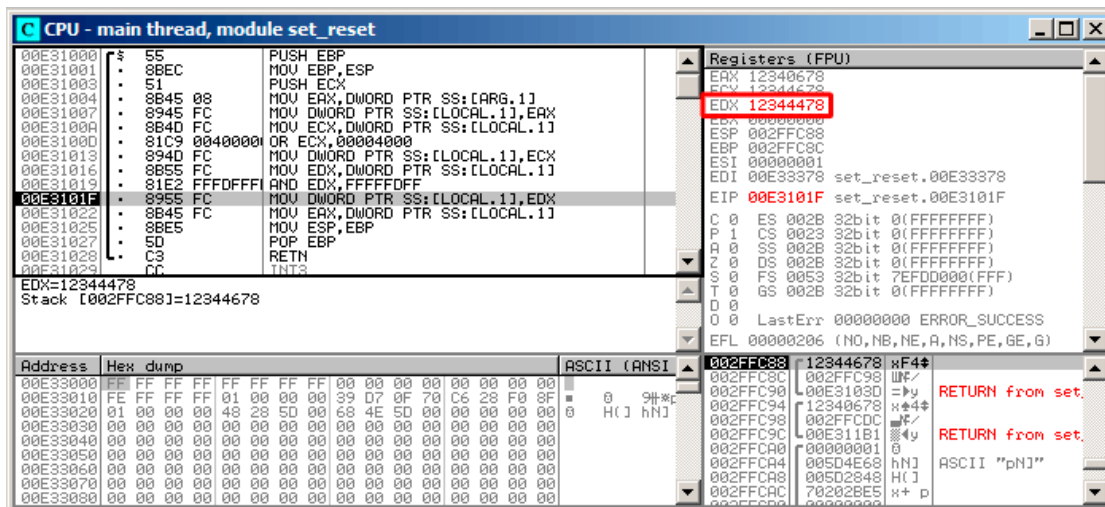


Рис. 1.97: OllyDbg: AND сработал

10-й бит очищен (или, иным языком, оставлены все биты кроме 10-го) и итоговое значение это 0x12344478 (0b100100011010001000100010001111000).

### Оптимизирующий MSVC

Если скомпилировать в MSVC с оптимизацией (/Ox), то код еще короче:

Листинг 1.276: Оптимизирующий MSVC

```

_a$ = 8 ; size = 4
_f PROC
    mov     eax, DWORD PTR _a$[esp-4]
    and     eax, -513 ; fffffffH
    or      eax, 16384 ; 00004000H
    ret     0
_f ENDP
  
```

### Неоптимизирующий GCC

Попробуем GCC 4.4.1 без оптимизации:

Листинг 1.277: Неоптимизирующий GCC

```

public f
f      proc near
  
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

var_4      = dword ptr -4
arg_0      = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 10h
                mov     eax, [ebp+arg_0]
                mov     [ebp+var_4], eax
                or      [ebp+var_4], 4000h
                and     [ebp+var_4], 0FFFFFFFh
                mov     eax, [ebp+var_4]
                leave
                retn
f            endp

```

Также избыточный код, хотя короче, чем у MSVC без оптимизации.

Попробуем теперь GCC с оптимизацией -O3:

### Оптимизирующий GCC

Листинг 1.278: Оптимизирующий GCC

```

f            public f
            proc near
arg_0      = dword ptr  8

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0]
                pop     ebp
                or      ah, 40h
                and     ah, 0FDh
                retn
f            endp

```

Уже короче. Важно отметить, что через регистр AH компилятор работает с частью регистра EAX. Это его часть от 8-го до 15-го бита включительно.

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
RAX <sup>x64</sup>							
EAX						AX	
						AH	AL

N.B. В 16-битном процессоре 8086 аккумулятор имел название AX и состоял из двух 8-битных половин — AL (младшая часть) и AH (старшая). В 80386 регистры были расширены до 32-бит, аккумулятор стал называться EAX, но в целях совместимости, к его *более старым* частям всё ещё можно обращаться как к AX/AH/AL.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Из-за того, что все x86 процессоры — наследники 16-битного 8086, эти *старые* 16-битные опкоды короче нежели более новые 32-битные. Поэтому инструкция `or ah, 40h` занимает только 3 байта. Было бы логичнее сгенерировать здесь `or eax, 04000h`, но это уже 5 байт, или даже 6 (если регистр в первом операнде не EAX).

### Оптимизирующий GCC и `regparm`

Если мы скомпилируем этот же пример не только с включенной оптимизацией -O3, но ещё и с опцией `regparm=3`, о которой я писал немного выше, то получится ещё короче:

Листинг 1.279: Оптимизирующий GCC

```
f      public f
      proc near
      push    ebp
      or     ah, 40h
      mov    ebp, esp
      and   ah, 0FDh
      pop    ebp
      retn
f      endp
```

Действительно — первый аргумент уже загружен в EAX, и прямо здесь можно начинать с ним работать. Интересно, что и пролог функции (`push ebp / mov ebp, esp`) и эпилог (`pop ebp`) функции можно смело выкинуть за ненадобностью, но возможно GCC ещё не так хорош для подобных оптимизаций по размеру кода. Впрочем, в реальной жизни подобные короткие функции лучше всего автоматически делать в виде *inline-функций* (?? (стр. ??)).

### ARM + Оптимизирующий Keil 6/2013 (Режим ARM)

Листинг 1.280: Оптимизирующий Keil 6/2013 (Режим ARM)

```
02 0C C0 E3      BIC    R0, R0, #0x200
01 09 80 E3      ORR    R0, R0, #0x4000
1E FF 2F E1      BX     LR
```

BIC (*Bitwise bit Clear*) это инструкция сбрасывающая заданные биты. Это как аналог AND, но только с инвертированным операндом.

Т.е. это аналог инструкций NOT +AND.

ORR это «логическое или», аналог OR в x86.

Пока всё понятно.

### ARM + Оптимизирующий Keil 6/2013 (Режим Thumb)

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Листинг 1.281: Оптимизирующий Keil 6/2013 (Режим Thumb)

01 21 89 03	MOVS	R1, 0x4000	
08 43	ORRS	R0, R1	
49 11	ASRS	R1, R1, #5	; сгенерировать 0x200 и записать в
88 43	BICS	R0, R1	
70 47	BX	LR	

Вероятно, Keil решил, что код в режиме Thumb, получающий 0x200 из 0x4000, более компактный, нежели код, записывающий 0x200 в какой-нибудь регистр.

Поэтому при помощи инструкции ASRS (арифметический сдвиг вправо), это значение вычисляется как  $0x4000 \gg 5$ .

### ARM + Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

Листинг 1.282: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

42 0C C0 E3	BIC	R0, R0, #0x4200
01 09 80 E3	ORR	R0, R0, #0x4000
1E FF 2F E1	BX	LR

Код, который был сгенерирован LLVM, в исходном коде, на самом деле, выглядел бы так:

```
REMOVE_BIT (rt, 0x4200);
SET_BIT (rt, 0x4000);
```

И он делает в точности что нам нужно. Но почему 0x4200? Возможно, это артефакт оптимизатора LLVM <sup>146</sup>. Возможно, ошибка оптимизатора компилятора, но создаваемый код всё же работает верно.

Об аномалиях компиляторов, подробнее читайте здесь (?? (стр. ??)).

Оптимизирующий Xcode 4.6.3 (LLVM) для режима Thumb генерирует точно такой же код.

### ARM: ещё об инструкции BIC

Если немного переделать пример:

```
int f(int a)
{
    int rt=a;

    REMOVE_BIT (rt, 0x1234);

    return rt;
};
```

То оптимизирующий Keil 5.03 в режиме ARM сделает такое:

<sup>146</sup>Это был LLVM build 2410.2.00 входящий в состав Apple Xcode 4.6.3



```
f PROC
    BIC    r0,r0,#0x1000
    BIC    r0,r0,#0x234
    BX     lr
    ENDP
```

Здесь две инструкции BIC, т.е. биты 0x1234 сбрасываются в два прохода.

Это потому что в инструкции BIC нельзя закодировать значение 0x1234, но можно закодировать 0x1000 либо 0x234.

### ARM64: Оптимизирующий GCC (Linaro) 4.9

Оптимизирующий GCC, компилирующий для ARM64, может использовать AND вместо BIC:

Листинг 1.283: Оптимизирующий GCC (Linaro) 4.9

```
f:
    and    w0, w0, -513    ; 0xFFFFFFFFFFFFDFF
    orr    w0, w0, 16384   ; 0x4000
    ret
```

### ARM64: Неоптимизирующий GCC (Linaro) 4.9

Неоптимизирующий GCC генерирует больше избыточного кода, но он работает также:

Листинг 1.284: Неоптимизирующий GCC (Linaro) 4.9

```
f:
    sub    sp, sp, #32
    str    w0, [sp,12]
    ldr    w0, [sp,12]
    str    w0, [sp,28]
    ldr    w0, [sp,28]
    orr    w0, w0, 16384   ; 0x4000
    str    w0, [sp,28]
    ldr    w0, [sp,28]
    and    w0, w0, -513    ; 0xFFFFFFFFFFFFDFF
    str    w0, [sp,28]
    ldr    w0, [sp,28]
    add    sp, sp, 32
    ret
```

## MIPS

Листинг 1.285: Оптимизирующий GCC 4.4.5 (IDA)

```
f:
; $a0=a
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

ori    $a0, 0x4000
; $a0=a|0x4000
li     $v0, 0xFFFFDFF
jr     $ra
and    $v0, $a0, $v0
; на выходе: $v0 = $a0 & $v0 = a|0x4000 & 0xFFFFDFF

```

ORI это, конечно, операция «ИЛИ», «|» в имени инструкции означает, что значение встроено в машинный код.

И напротив, есть AND. Здесь нет возможности использовать ANDI, потому что невозможно встроить число 0xFFFFDFF в одну инструкцию, так что компилятору приходится в начале загружать значение 0xFFFFDFF в регистр \$V0, а затем генерировать AND, которая возьмет все значения из регистров.

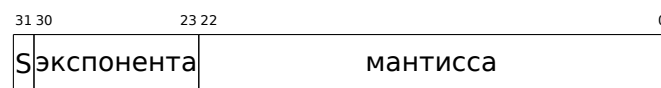
### 1.28.3. Сдвиги

Битовые сдвиги в Си/Си++ реализованы при помощи операторов «<<» и «>>». В x86 есть инструкции SHL (SHift Left) и SHR (SHift Right) для этого. Инструкции сдвига также активно применяются при делении или умножении на числа-степени двойки:  $2^n$  (т.е. 1, 2, 4, 8, и т. д.): ?? (стр. ??), ?? (стр. ??).

Операции сдвига ещё потому так важны, потому что они часто используются для изолирования определенного бита или для конструирования значения из нескольких разрозненных бит.

### 1.28.4. Установка и сброс отдельного бита: пример с FPU!

Как мы уже можем знать, вот как биты расположены в значении типа *float* в формате IEEE 754:



( S — знак )

Знак числа — это **MSB!**<sup>147</sup>. Возможно ли работать со знаком числа с плавающей точкой, не используя FPU-инструкций?

```

#include <stdio.h>

float my_abs (float i)
{
    unsigned int tmp=*(unsigned int*)&i & 0x7FFFFFFF;
    return *(float*)&tmp;
};

float set_sign (float i)
{
    unsigned int tmp=*(unsigned int*)&i | 0x80000000;

```

<sup>147</sup> **MSB!**

```

        return *(float*)&tmp;
};

float negate (float i)
{
    unsigned int tmp=*(unsigned int*)&i ^ 0x80000000;
    return *(float*)&tmp;
};

int main()
{
    printf ("my_abs():\n");
    printf ("%f\n", my_abs (123.456));
    printf ("%f\n", my_abs (-456.123));
    printf ("set_sign():\n");
    printf ("%f\n", set_sign (123.456));
    printf ("%f\n", set_sign (-456.123));
    printf ("negate():\n");
    printf ("%f\n", negate (123.456));
    printf ("%f\n", negate (-456.123));
};

```

Придется использовать эти трюки в Си/Си++ с типами данных чтобы копировать из значения типа *float* и обратно без конверсии. Так что здесь три функции: *my\_abs()* сбрасывает **MSB!**; *set\_sign()* устанавливает **MSB!** и *negate()* меняет его на противоположный.

XOR может использоваться для смены бита.

## x86

Код прямолинеен:

Листинг 1.286: Оптимизирующий MSVC 2012

```

_tmp$ = 8
_i$ = 8
_my_abs PROC
    and     DWORD PTR _i$[esp-4], 2147483647 ; 7fffffffH
    fld     DWORD PTR _tmp$[esp-4]
    ret     0
_my_abs ENDP

_tmp$ = 8
_i$ = 8
_set_sign PROC
    or      DWORD PTR _i$[esp-4], -2147483648 ; 80000000H
    fld     DWORD PTR _tmp$[esp-4]
    ret     0
_set_sign ENDP

_tmp$ = 8
_i$ = 8

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

_negate PROC
    xor     DWORD PTR _i$[esp-4], -2147483648 ; 80000000H
    fld     DWORD PTR _tmp$[esp-4]
    ret     0
_negate ENDP

```

Входное значение типа *float* берется из стека, но мы обходимся с ним как с целочисленным значением.

AND и OR сбрасывают и устанавливают нужный бит. XOR переворачивает его.

В конце измененное значение загружается в ST0, потому что числа с плавающей точкой возвращаются в этом регистре.

Попробуем оптимизирующий MSVC 2012 для x64:

Листинг 1.287: Оптимизирующий MSVC 2012 x64

```

tmp$ = 8
i$ = 8
my_abs PROC
    movss  DWORD PTR [rsp+8], xmm0
    mov    eax, DWORD PTR i$[rsp]
    btr   eax, 31
    mov    DWORD PTR tmp$[rsp], eax
    movss  xmm0, DWORD PTR tmp$[rsp]
    ret    0
my_abs ENDP
_TEXT   ENDS

tmp$ = 8
i$ = 8
set_sign PROC
    movss  DWORD PTR [rsp+8], xmm0
    mov    eax, DWORD PTR i$[rsp]
    bts   eax, 31
    mov    DWORD PTR tmp$[rsp], eax
    movss  xmm0, DWORD PTR tmp$[rsp]
    ret    0
set_sign ENDP

tmp$ = 8
i$ = 8
negate PROC
    movss  DWORD PTR [rsp+8], xmm0
    mov    eax, DWORD PTR i$[rsp]
    btc   eax, 31
    mov    DWORD PTR tmp$[rsp], eax
    movss  xmm0, DWORD PTR tmp$[rsp]
    ret    0
negate ENDP

```

Во-первых, входное значение передается в XMM0, затем оно копируется в локальный стек и затем мы видим новые для нас инструкции: BTR, BTS, BTC. Эти

инструкции используются для сброса определенного бита (BTR: «reset»), установки (BTS: «set») и инвертирования (BTC: «complement»). 31-й бит это **MSB!**, если считать начиная с нуля. И наконец, результат копируется в регистр XMM0, потому что значения с плавающей точкой возвращаются в регистре XMM0 в среде Win64.

## MIPS

GCC 4.4.5 для MIPS делает почти то же самое:

Листинг 1.288: Оптимизирующий GCC 4.4.5 (IDA)

```

my_abs:
; скопировать из сопроцессора 1:
    mfc1    $v1, $f12
    li      $v0, 0x7FFFFFFF
; $v0=0x7FFFFFFF
; применить И:
    and     $v0, $v1
; скопировать в сопроцессор 1:
    mtc1    $v0, $f0
; возврат
    jr      $ra
    or      $at, $zero ; branch delay slot

set_sign:
; скопировать из сопроцессора 1:
    mfc1    $v0, $f12
    lui     $v1, 0x8000
; $v1=0x80000000
; применить ИЛИ:
    or      $v0, $v1, $v0
; скопировать в сопроцессор 1:
    mtc1    $v0, $f0
; возврат
    jr      $ra
    or      $at, $zero ; branch delay slot

negate:
; скопировать из сопроцессора 1:
    mfc1    $v0, $f12
    lui     $v1, 0x8000
; $v1=0x80000000
; применить исключающее ИЛИ:
    xor     $v0, $v1, $v0
; скопировать в сопроцессор 1:
    mtc1    $v0, $f0
; возврат
    jr      $ra
    or      $at, $zero ; branch delay slot

```

Для загрузки константы 0x80000000 в регистр используется только одна инструкция LUI, потому что LUI сбрасывает младшие 16 бит и это нули в констан-

те, так что одной LUI без ORI достаточно.

## ARM

### Оптимизирующий Keil 6/2013 (Режим ARM)

Листинг 1.289: Оптимизирующий Keil 6/2013 (Режим ARM)

```

my_abs PROC
; очистить бит:
    BIC    r0,r0,#0x80000000
    BX     lr
    ENDP

set_sign PROC
; применить ИЛИ:
    ORR    r0,r0,#0x80000000
    BX     lr
    ENDP

negate PROC
; применить исключающее ИЛИ:
    EOR    r0,r0,#0x80000000
    BX     lr
    ENDP

```

Пока всё понятно. В ARM есть инструкция BIC для сброса заданных бит. EOR это инструкция в ARM которая делает то же что и XOR («Exclusive OR»).

### Оптимизирующий Keil 6/2013 (Режим Thumb)

Листинг 1.290: Оптимизирующий Keil 6/2013 (Режим Thumb)

```

my_abs PROC
    LSLS   r0,r0,#1
; r0=i<<1
    LSRS   r0,r0,#1
; r0=(i<<1)>>1
    BX     lr
    ENDP

set_sign PROC
    MOVS   r1,#1
; r1=1
    LSLS   r1,r1,#31
; r1=1<<31=0x80000000
    ORRS   r0,r0,r1
; r0=r0 | 0x80000000
    BX     lr
    ENDP

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
negate PROC
    MOVS    r1,#1
; r1=1
    LSLS    r1,r1,#31
; r1=1<<31=0x80000000
    EORS    r0,r0,r1
; r0=r0 ^ 0x80000000
    BX      lr
    ENDP
```

В режиме Thumb 16-битные инструкции, в которых нельзя задать много данных, поэтому здесь применяется пара инструкций MOVS/LSLS для формирования константы 0x80000000.

Это работает как выражение:  $1 \ll 31 = 0x80000000$ .

Код `my_abs` выглядит странно и работает как выражение:  $(i \ll 1) \gg 1$ . Это выражение выглядит бессмысленным. Но тем не менее, когда исполняется `input << 1`, **MSB!** (бит знака) просто выбрасывается. Когда исполняется следующее выражение `result >> 1`, все биты становятся на свои места, а **MSB!** ноль, потому что все «новые» биты появляющиеся во время операций сдвига это всегда нули. Таким образом, пара инструкций LSLS/LSRS сбрасывают **MSB!**.

### Оптимизирующий GCC 4.6.3 (Raspberry Pi, Режим ARM)

Листинг 1.291: Оптимизирующий GCC 4.6.3 для Raspberry Pi (Режим ARM)

```
my_abs
; скопировать из S0 в R2:
    FMRS    R2, S0
; очистить бит:
    BIC     R3, R2, #0x80000000
; скопировать из R3 в S0:
    FMSR    S0, R3
    BX      LR

set_sign
; скопировать из S0 в R2:
    FMRS    R2, S0
; применить ИЛИ:
    ORR     R3, R2, #0x80000000
; скопировать из R3 в S0:
    FMSR    S0, R3
    BX      LR

negate
; скопировать из S0 в R2:
    FMRS    R2, S0
; применить операцию сложения:
    ADD     R3, R2, #0x80000000
; скопировать из R3 в S0:
    FMSR    S0, R3
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мой адреса](#). Спасибо!

Запустим Raspberry Pi Linux в QEMU и он эмулирует FPU в ARM, так что здесь используются S-регистры для передачи значений с плавающей точкой вместо R-регистров.

Инструкция FMRS копирует данные из **GPR!** в FPU и назад. `my_abs()` и `set_sign()` выглядят предсказуемо, но `negate()`? Почему там ADD вместо XOR?

Трудно поверить, но инструкция `ADD register, 0x80000000` работает так же как и

`XOR register, 0x80000000`. Прежде всего, какая наша цель? Цель в том, чтобы поменять **MSB!** на противоположный, и давайте забудем пока об операции XOR.

Из школьной математики мы можем помнить, что прибавление числа вроде 1000 к другому никогда не затрагивает последние 3 цифры.

Например:  $1234567 + 10000 = 1244567$  (последние 4 цифры никогда не меняются). Но мы работаем с двоичной системой исчисления, и `0x80000000` это `0b10000000000000000000000000000000` в двоичной системе, т.е. только старший бит установлен.

Прибавление `0x80000000` к любому значению никогда не затронет младших 31 бит, а только **MSB!**

Прибавление 1 к 0 в итоге даст 1. Прибавление 1 к 1 даст `0b10` в двоичном виде, но 32-й бит (считая с нуля) выброшен, потому что наши регистры имеют ширину в 32 бита. Так что результат — 0.

Вот почему XOR здесь можно заменить на ADD. Трудно сказать, почему GCC решил сделать так, но это работает корректно.

### 1.28.5. Подсчет выставленных бит

Вот этот несложный пример иллюстрирует функцию, считающую количество бит-единиц во входном значении.

Эта операция также называется «population count»<sup>148</sup>.

```
#include <stdio.h>

#define IS_SET(flag, bit)      ((flag) & (bit))

int f(unsigned int a)
{
    int i;
    int rt=0;

    for (i=0; i<32; i++)
        if (IS_SET (a, 1<<i))
            rt++;
}
```

<sup>148</sup> современные x86-процессоры (поддерживающие SSE4) даже имеют инструкцию POPCNT для этого



```

    return rt;
};

int main()
{
    f(0x12345678); // test
};

```

В этом цикле счетчик итераций  $i$  считает от 0 до 31, а  $1 \ll i$  будет от 1 до  $0x80000000$ . Описывая это словами, можно сказать *сдвинуть единицу на  $n$  бит влево*. Т.е. в некотором смысле, выражение  $1 \ll i$  последовательно выдает все возможные позиции бит в 32-битном числе. Освободившийся бит справа всегда обнуляется.

Вот таблица всех возможных значений  $1 \ll i$  для  $i = 0 \dots 31$ :

Выражение	Степень двойки	Десятичная форма	Шестнадцатеричная
$1 \ll 0$	$2^0$	1	1
$1 \ll 1$	$2^1$	2	2
$1 \ll 2$	$2^2$	4	4
$1 \ll 3$	$2^3$	8	8
$1 \ll 4$	$2^4$	16	0x10
$1 \ll 5$	$2^5$	32	0x20
$1 \ll 6$	$2^6$	64	0x40
$1 \ll 7$	$2^7$	128	0x80
$1 \ll 8$	$2^8$	256	0x100
$1 \ll 9$	$2^9$	512	0x200
$1 \ll 10$	$2^{10}$	1024	0x400
$1 \ll 11$	$2^{11}$	2048	0x800
$1 \ll 12$	$2^{12}$	4096	0x1000
$1 \ll 13$	$2^{13}$	8192	0x2000
$1 \ll 14$	$2^{14}$	16384	0x4000
$1 \ll 15$	$2^{15}$	32768	0x8000
$1 \ll 16$	$2^{16}$	65536	0x10000
$1 \ll 17$	$2^{17}$	131072	0x20000
$1 \ll 18$	$2^{18}$	262144	0x40000
$1 \ll 19$	$2^{19}$	524288	0x80000
$1 \ll 20$	$2^{20}$	1048576	0x100000
$1 \ll 21$	$2^{21}$	2097152	0x200000
$1 \ll 22$	$2^{22}$	4194304	0x400000
$1 \ll 23$	$2^{23}$	8388608	0x800000
$1 \ll 24$	$2^{24}$	16777216	0x1000000
$1 \ll 25$	$2^{25}$	33554432	0x2000000
$1 \ll 26$	$2^{26}$	67108864	0x4000000
$1 \ll 27$	$2^{27}$	134217728	0x8000000
$1 \ll 28$	$2^{28}$	268435456	0x10000000
$1 \ll 29$	$2^{29}$	536870912	0x20000000
$1 \ll 30$	$2^{30}$	1073741824	0x40000000
$1 \ll 31$	$2^{31}$	2147483648	0x80000000

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Это числа-константы (битовые маски), которые крайне часто попадают в практику reverse engineer-а, и их нужно уметь распознавать.

Числа в десятичном виде, до 65536 и числа в шестнадцатеричном виде легко запомнить и так. А числа в десятичном виде после 65536, пожалуй, заучивать не нужно.

Эти константы очень часто используются для определения отдельных бит как флагов.

Например, это из файла `ssl_private.h` из исходников Apache 2.4.6:

```
/**
 * Define the SSL options
 */
#define SSL_OPT_NONE          (0)
#define SSL_OPT_RELSET       (1<<0)
#define SSL_OPT_STDENVVARS   (1<<1)
#define SSL_OPT_EXPORTCERTDATA (1<<3)
#define SSL_OPT_FAKEBASICAUTH (1<<4)
#define SSL_OPT_STRICTREQUIRE (1<<5)
#define SSL_OPT_OPTRENEGOTIATE (1<<6)
#define SSL_OPT_LEGACYDNFORMAT (1<<7)
```

Вернемся назад к нашему примеру.

Макрос `IS_SET` проверяет наличие этого бита в *a*.

Макрос `IS_SET` на самом деле это операция логического И (*AND*) и она возвращает 0 если бита там нет, либо эту же битовую маску, если бит там есть. В Си/Си++, конструкция `if()` срабатывает, если выражение внутри её не ноль, пусть хоть 123456, поэтому все будет работать.

## x86

### MSVC

Компилируем (MSVC 2010):

Листинг 1.292: MSVC 2010

```
_rt$ = -8          ; size = 4
_i$ = -4          ; size = 4
_a$ = 8           ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 8
    mov     DWORD PTR _rt$[ebp], 0
    mov     DWORD PTR _i$[ebp], 0
    jmp     SHORT $LN4@f
$LN3@f:
    mov     eax, DWORD PTR _i$[ebp] ; инкремент i
    add     eax, 1
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov     DWORD PTR _i$[ebp], eax
$LN4@f:
cmp     DWORD PTR _i$[ebp], 32      ; 00000020H
jge     SHORT $LN2@f                ; цикл закончился?
mov     edx, 1
mov     ecx, DWORD PTR _i$[ebp]
shl     edx, cl                     ; EDX=EDX<<CL
and     edx, DWORD PTR _a$[ebp]
je      SHORT $LN1@f                ; результат исполнения инструкции AND
был 0?
mov     eax, DWORD PTR _rt$[ebp]    ; тогда пропускаем следующие команды
add     eax, 1                      ; нет, не ноль
mov     DWORD PTR _rt$[ebp], eax   ; инкремент rt
$LN1@f:
jmp     SHORT $LN3@f
$LN2@f:
mov     eax, DWORD PTR _rt$[ebp]
mov     esp, ebp
pop     ebp
ret     0
_f     ENDP

```

## OllyDbg

Загрузим этот пример в OllyDbg. Входное значение для функции пусть будет 0x12345678.

Для  $i = 1$ , мы видим, как  $i$  загружается в ECX:

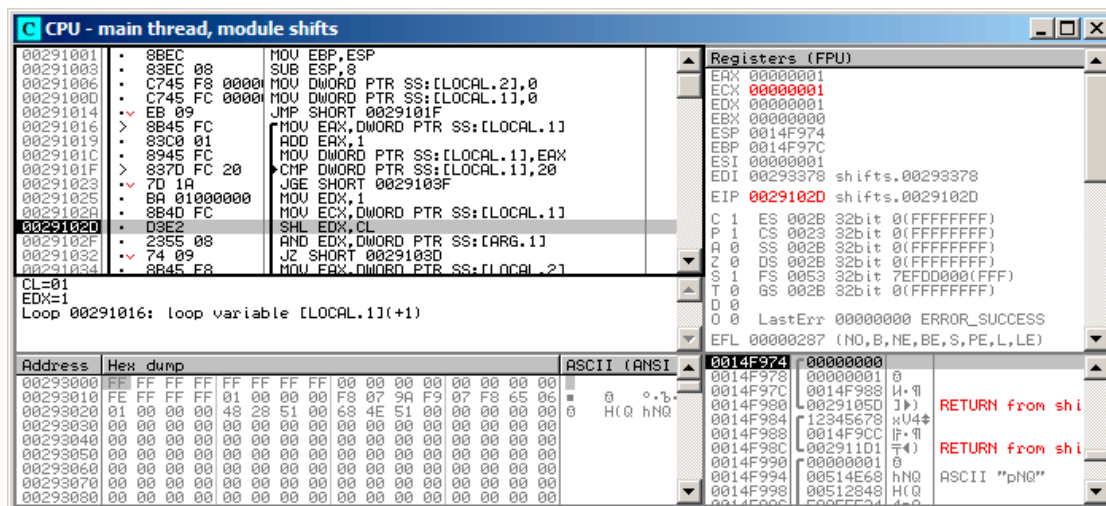


Рис. 1.98: OllyDbg:  $i = 1$ ,  $i$  загружено в ECX

EDX содержит 1. Сейчас будет исполнена SHL.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

SHL исполнилась:

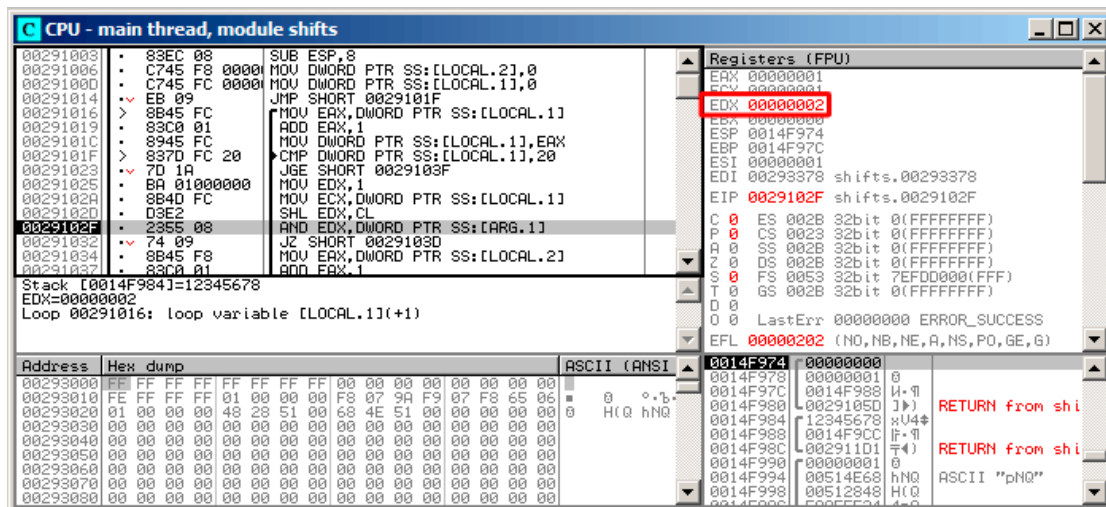


Рис. 1.99: OlyDbg:  $i = 1$ ,  $EDX = 1 \ll 1 = 2$

EDX содержит  $1 \ll 1$  (или 2). Это битовая маска.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

AND устанавливает ZF в 1, что означает, что входное значение (0x12345678) умножается<sup>149</sup> с 2 давая в результате 0:

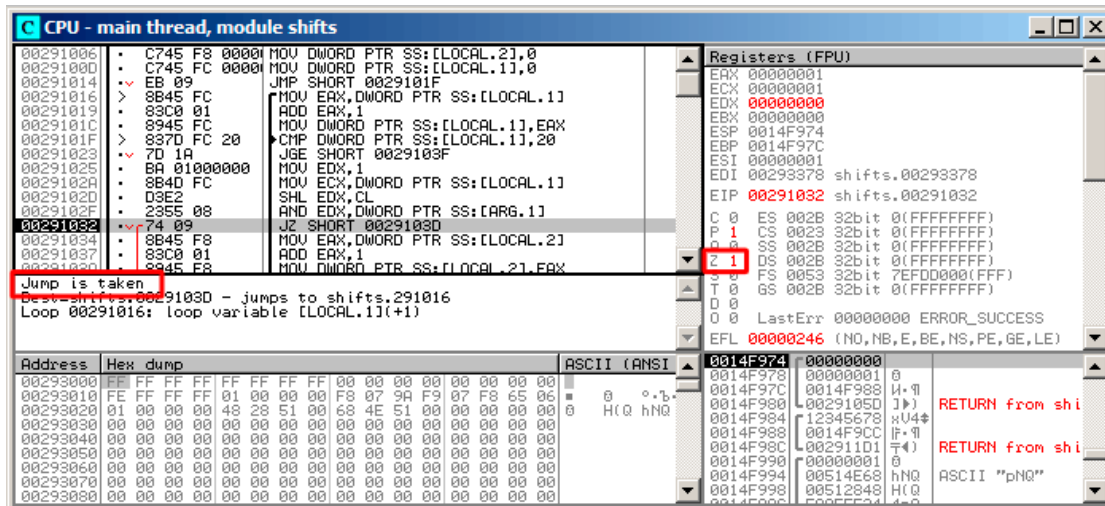


Рис. 1.100: OllyDbg:  $i = 1$ , есть ли этот бит во входном значении? Нет. (ZF =1)

Так что во входном значении соответствующего бита нет. Участок кода, увеличивающий счетчик бит на единицу, не будет исполнен: инструкция JZ обойдет его.

<sup>149</sup> Логическое «И»

Немного потрассируем далее и *i* теперь 4.

SHL исполнилась:

**CPU - main thread, module shifts**

Address	Hex dump	ASCII (ANSI)
00293000	FF FF FF FF FF FF FF FF 00 00 00 00 00 00 00 00	
00293010	FE FF FF FF 01 00 00 00 F8 07 3A F9 07 F8 65 06	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
00293020	01 00 00 00 43 23 51 00 68 4E 51 00 00 00 00 00	H(Q hNQ
00293030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00293040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00293050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00293060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00293070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00293080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

**Registers (FPU)**

EAX	00000004
ECX	00000004
EDX	00000001
EBX	00000000
ESP	0014F974
EBP	0014F97C
ESI	00000001
EDI	00293378 shifts.00293378
EIP	0029102D shifts.0029102D
C 1	ES 002B 32bit 0(FFFFFFFF)
P 1	CS 0023 32bit 0(FFFFFFFF)
A 0	SS 002B 32bit 0(FFFFFFFF)
Z 0	DS 002B 32bit 0(FFFFFFFF)
S 1	FS 0053 32bit 7EFD0000(FFF)
T 0	GS 002B 32bit 0(FFFFFFFF)
D 0	
O 0	LastErr 00000000 ERROR_SUCCESS
EFL	00000287 (NO,B,NE,BE,S,PE,L,LE)

CL=04  
EDX=1  
Loop 00291016: loop variable [LOCAL.1](+1)

Рис. 1.101: OllyDbg:  $i = 4$ ,  $i$  загружено в ECX

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

EDX = 1 << 4 (или 0x10 или 16):

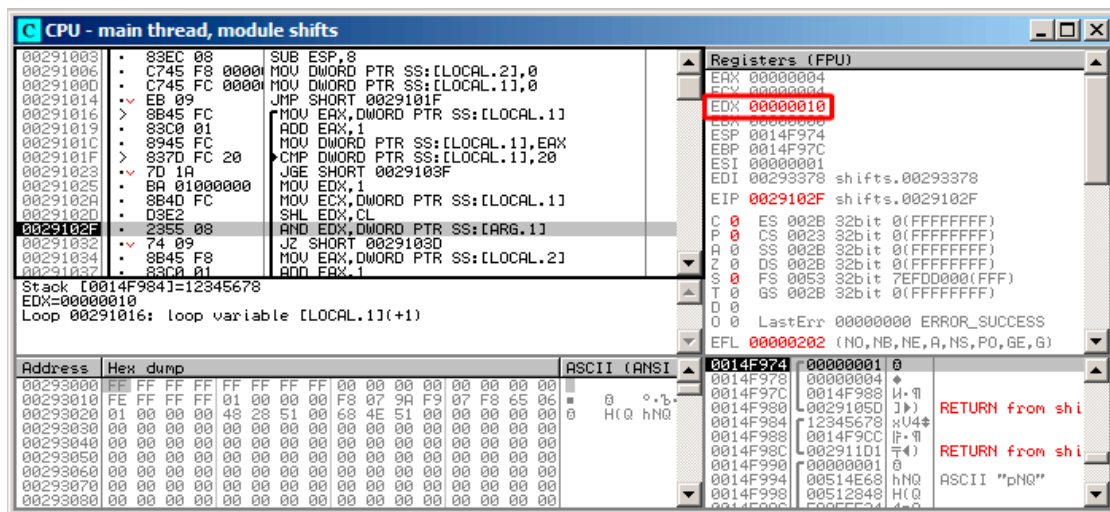


Рис. 1.102: OllyDbg:  $i = 4$ ,  $EDX = 1 \ll 4 = 0x10$

Это ещё одна битовая маска.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



AND исполнилась:

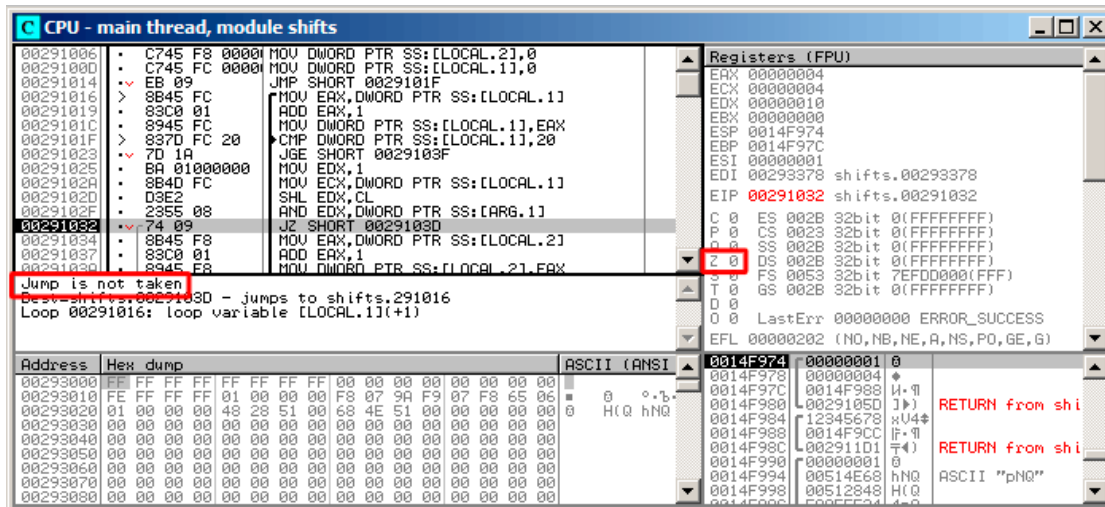


Рис. 1.103: OllyDbg:  $i = 4$ , есть ли этот бит во входном значении? Да. (ZF = 0)

ZF сейчас 0 потому что этот бит присутствует во входном значении. Действительно,  $0x12345678 \& 0x10 = 0x10$ . Этот бит считается: переход не сработает и счетчик бит будет увеличен на единицу.

Функция возвращает 13. Это количество установленных бит в значении  $0x12345678$ .

## GCC

Скомпилируем то же и в GCC 4.4.1:

Листинг 1.293: GCC 4.4.1

```

public f
f
proc near

rt
    = dword ptr -0Ch
i
    = dword ptr -8
arg_0
    = dword ptr 8

    push    ebp
    mov     ebp, esp
    push    ebx
    sub     esp, 10h
    mov     [ebp+rt], 0
    mov     [ebp+i], 0
    jmp     short loc_80483EF

loc_80483D0:
    mov     eax, [ebp+i]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        mov     edx, 1
        mov     ebx, edx
        mov     ecx, eax
        shl     ebx, cl
        mov     eax, ebx
        and     eax, [ebp+arg_0]
        test    eax, eax
        jz     short loc_80483EB
        add     [ebp+rt], 1
loc_80483EB:
        add     [ebp+i], 1
loc_80483EF:
        cmp     [ebp+i], 1Fh
        jle    short loc_80483D0
        mov     eax, [ebp+rt]
        add     esp, 10h
        pop     ebx
        pop     ebp
        retn
f      endp

```

## x64

Немного изменим пример, расширив его до 64-х бит:

```

#include <stdio.h>
#include <stdint.h>

#define IS_SET(flag, bit)      ((flag) & (bit))

int f(uint64_t a)
{
    uint64_t i;
    int rt=0;

    for (i=0; i<64; i++)
        if (IS_SET (a, 1ULL<<i))
            rt++;

    return rt;
};

```

## Неоптимизирующий GCC 4.8.2

Пока всё просто.

Листинг 1.294: Неоптимизирующий GCC 4.8.2

```

f:
    push    rbp
    mov     rbp, rsp

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        mov     QWORD PTR [rbp-24], rdi ; a
        mov     DWORD PTR [rbp-12], 0  ; rt=0
        mov     QWORD PTR [rbp-8], 0   ; i=0
        jmp     .L2
.L4:
        mov     rax, QWORD PTR [rbp-8]
        mov     rdx, QWORD PTR [rbp-24]
; RAX = i, RDX = a
        mov     ecx, eax
; ECX = i
        shr     rdx, cl
; RDX = RDX>>CL = a>>i
        mov     rax, rdx
; RAX = RDX = a>>i
        and     eax, 1
; EAX = EAX&1 = (a>>i)&1
        test    rax, rax
; последний бит был нулевым?
; пропустить следующую инструкцию ADD, если это было так.
        je     .L3
        add     DWORD PTR [rbp-12], 1  ; rt++
.L3:
        add     QWORD PTR [rbp-8], 1   ; i++
.L2:
        cmp     QWORD PTR [rbp-8], 63 ; i<63?
        jbe    .L4                      ; перейти на начало тела цикла, если
это так
        mov     eax, DWORD PTR [rbp-12] ; возврат rt
        pop     rbp
        ret

```

## Оптимизирующий GCC 4.8.2

Листинг 1.295: Оптимизирующий GCC 4.8.2

```

1 f:
2     xor     eax, eax           ; переменная rt будет находиться в регистре
3     EAX   xor     ecx, ecx           ; переменная i будет находиться в регистре
4     ECX
5     .L3:   mov     rsi, rdi         ; загрузить входное значение
6           lea     edx, [rax+1]     ; EDX=EAX+1
7           ; EDX здесь это новая версия rt,
8           ; которая будет записана в переменную rt, если последний бит был 1
9           shr     rsi, cl          ; RSI=RSI>>CL
10          and     esi, 1           ; ESI=ESI&1
11          ; последний бит был 1? Тогда записываем новую версию rt в EAX
12          cmovne eax, edx
13          add     rcx, 1           ; RCX++
14          cmp     rcx, 64
15          jne    .L3
16          rep ret                 ; AKA fatret

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Код более лаконичный, но содержит одну необычную вещь. Во всех примерах, что мы пока видели, инкремент значения переменной «rt» происходит после сравнения определенного бита с единицей, но здесь «rt» увеличивается на 1 до этого (строка б), записывая новое значение в регистр EDX.

Затем, если последний бит был 1, инструкция `CMOVNE`<sup>150</sup> (которая синонимична `CMOVNZ`<sup>151</sup>) фиксирует новое значение «rt» копируя значение из EDX («предполагаемое значение rt») в EAX («текущее rt» которое будет возвращено в конце функции). Следовательно, инкремент происходит на каждом шаге цикла, т.е. 64 раза, вне всякой связи с входным значением.

Преимущество этого кода в том, что он содержит только один условный переход (в конце цикла) вместо двух (пропускающий инкремент «rt» и ещё одного в конце цикла).

И это может работать быстрее на современных CPU с предсказателем переходов: ?? (стр. ??).

Последняя инструкция это `REP RET` (опкод F3 C3) которая также называется `FATRET` в MSVC. Это оптимизированная версия `RET`, рекомендуемая AMD для вставки в конце функции, если `RET` идет сразу после условного перехода: [Software Optimization Guide for AMD Family 16h Processors, (2013)p.15]<sup>152</sup>.

## Оптимизирующий MSVC 2010

Листинг 1.296: Оптимизирующий MSVC 2010

```

a$ = 8
f      PROC
; RCX = входное значение
      xor     eax, eax
      mov     edx, 1
      lea    r8d, QWORD PTR [rax+64]
; R8D=64
      npad   5
$LL4@f:
      test   rdx, rcx
; не было такого бита во входном значении?
; тогда пропустить следующую инструкцию INC.
      je     SHORT $LN3@f
      inc    eax      ; rt++
$LN3@f:
      rol    rdx, 1   ; RDX=RDX<<1
      dec    r8       ; R8--
      jne   SHORT $LL4@f
      fatret 0
f      ENDP

```

<sup>150</sup>Conditional MOVE if Not Equal (MOV если не равно)

<sup>151</sup>Conditional MOVE if Not Zero (MOV если не ноль)

<sup>152</sup>Больше об этом: <http://repzret.org/p/repzret/>

Здесь используется инструкция ROL вместо SHL, которая на самом деле «rotate left» (прокручивать влево) вместо «shift left» (сдвиг влево), но здесь, в этом примере, она работает так же как и SHL.

Об этих «прокручивающих» инструкциях больше читайте здесь: ?? (стр. ??).

R8 здесь считает от 64 до 0. Это как бы инвертированная переменная *i*.

Вот таблица некоторых регистров в процессе исполнения:

RDX	R8
0x0000000000000001	64
0x0000000000000002	63
0x0000000000000004	62
0x0000000000000008	61
...	...
0x4000000000000000	2
0x8000000000000000	1

В конце видим инструкцию FATRET, которая была описана здесь: ?? (стр. ??).

## Оптимизирующий MSVC 2012

Листинг 1.297: Оптимизирующий MSVC 2012

```

a$ = 8
f PROC
; RCX = входное значение
xor    eax, eax
mov    edx, 1
lea    r8d, QWORD PTR [rax+32]
; EDX = 1, R8D = 32
npad   5
$LL4@f:
; проход 1 -----
test   rdx, rcx
je     SHORT $LN3@f
inc    eax    ; rt++
$LN3@f:
rol    rdx, 1 ; RDX=RDX<<1
; -----
; проход 2 -----
test   rdx, rcx
je     SHORT $LN11@f
inc    eax    ; rt++
$LN11@f:
rol    rdx, 1 ; RDX=RDX<<1
; -----
dec    r8     ; R8--
jne    SHORT $LL4@f
fatret 0
f ENDP

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Оптимизирующий MSVC 2012 делает почти то же самое что и оптимизирующий MSVC 2010, но почему-то он генерирует 2 идентичных тела цикла и счетчик цикла теперь 32 вместо 64. Честно говоря, нельзя сказать, почему. Какой-то трюк с оптимизацией? Может быть, телу цикла лучше быть немного длиннее?

Так или иначе, такой код здесь уместен, чтобы показать, что результат компилятора иногда может быть очень странный и нелогичный, но прекрасно работающий, конечно же.

### ARM + Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

Листинг 1.298: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

	MOV	R1, R0	
	MOV	R0, #0	
	MOV	R2, #1	
	MOV	R3, R0	
loc_2E54	TST	R1, R2, LSL R3 ; установить флаги в соответствии с R1 & (R2<<R3)	
	ADD	R3, R3, #1 ; R3++	
	ADDNE	R0, R0, #1 ; если флаг ZF сброшен TST, то R0++	
	CMP	R3, #32	
	BNE	loc_2E54	
	BX	LR	

TST это то же что и TEST в x86.

Как уже было указано (?? (стр. ??)), в режиме ARM нет отдельной инструкции для сдвигов.

Однако, модификаторами LSL (*Logical Shift Left*), LSR (*Logical Shift Right*), ASR (*Arithmetic Shift Right*), ROR (*Rotate Right*) и RRX (*Rotate Right with Extend*) можно дополнять некоторые инструкции, такие как MOV, TST, CMP, ADD, SUB, RSB<sup>153</sup>.

Эти модификаторы указывают, как сдвигать второй операнд, и на сколько.

Таким образом, инструкция «TST R1, R2, LSL R3» здесь работает как  $R1 \wedge (R2 \ll R3)$ .

### ARM + Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

Почти такое же, только здесь применяется пара инструкций LSL.W/TST вместо одной TST, ведь в режиме Thumb нельзя добавлять модификатор LSL прямо в TST.

	MOV	R1, R0	
	MOVS	R0, #0	
	MOV.W	R9, #1	
	MOVS	R3, #0	
loc_2F7A			

<sup>153</sup>Эти инструкции также называются «data processing instructions»

LSL.W	R2, R9, R3
TST	R2, R1
ADD.W	R3, R3, #1
IT NE	
ADDNE	R0, #1
CMP	R3, #32
BNE	loc_2F7A
BX	LR

## ARM64 + Оптимизирующий GCC 4.9

Возьмем 64-битный пример, который уже был здесь использован: ?? (стр. ??).

Листинг 1.299: Оптимизирующий GCC (Linaro) 4.8

```
f:
    mov    w2, 0           ; rt=0
    mov    x5, 1
    mov    w1, w2
.L2:
    lsl    x4, x5, x1      ; w4 = w5<<w1 = 1<<i
    add    w3, w2, 1      ; new_rt=rt+1
    tst    x4, x0         ; (1<<i) & a
    add    w1, w1, 1      ; i++
; результат TST был ненулевой?
; тогда w2=w3 или rt=new_rt.
; в противном случае: w2=w2 или rt=rt (холостая операция)
    csel   w2, w3, w2, ne
    cmp    w1, 64         ; i<64?
    bne   .L2            ; да
    mov    w0, w2        ; возврат rt
    ret
```

Результат очень похож на тот, что GCC сгенерировал для x64: ?? (стр. ??).

Инструкция CSEL это «Conditional SElect» (выбор при условии). Она просто выбирает одну из переменных, в зависимости от флагов выставленных TST и копирует значение в регистр W2, содержащий переменную «rt».

## ARM64 + Неоптимизирующий GCC 4.9

И снова будем использовать 64-битный пример, который мы использовали ранее: ?? (стр. ??). Код более многословный, как обычно.

Листинг 1.300: Неоптимизирующий GCC (Linaro) 4.8

```
f:
    sub    sp, sp, #32
    str    x0, [sp,8]     ; сохранить значение "a" в Register Save Area
    str    wzr, [sp,24]   ; rt=0
    str    wzr, [sp,28]   ; i=0
    b     .L4
.L4:
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        ldr    w0, [sp,28]
        mov    x1, 1
        lsl    x0, x1, x0      ; X0 = X1<<X0 = 1<<i
        mov    x1, x0
; X1 = 1<<i
        ldr    x0, [sp,8]
; X0 = a
        and    x0, x1, x0
; X0 = X1&X0 = (1<<i) & a
; X0 содержит ноль? тогда перейти на .L3, пропуская инкремент "rt"
        cmp    x0, xzr
        beq    .L3
; rt++
        ldr    w0, [sp,24]
        add    w0, w0, 1
        str    w0, [sp,24]
.L3:
; i++
        ldr    w0, [sp,28]
        add    w0, w0, 1
        str    w0, [sp,28]
.L2:
; i<=63? тогда перейти на .L4
        ldr    w0, [sp,28]
        cmp    w0, 63
        ble    .L4
; возврат rt
        ldr    w0, [sp,24]
        add    sp, sp, 32
        ret

```

## MIPS

### Неоптимизирующий GCC

Листинг 1.301: Неоптимизирующий GCC 4.4.5 (IDA)

```

f:
; IDA не знает об именах переменных, мы присвоили их вручную:
rt      = -0x10
i       = -0xC
var_4   = -4
a       = 0

        addiu  $sp, -0x18
        sw     $fp, 0x18+var_4($sp)
        move   $fp, $sp
        sw     $a0, 0x18+a($fp)
; инициализировать переменные rt и i в ноль:
        sw     $zero, 0x18+rt($fp)
        sw     $zero, 0x18+i($fp)
; перейти на инструкции проверки цикла:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

        b        loc_68
        or        $at, $zero ; branch delay slot, NOP

loc_20:
        li        $v1, 1
        lw        $v0, 0x18+i($fp)
        or        $at, $zero ; load delay slot, NOP
        sllv     $v0, $v1, $v0
; $v0 = 1<<i
        move     $v1, $v0
        lw        $v0, 0x18+a($fp)
        or        $at, $zero ; load delay slot, NOP
        and      $v0, $v1, $v0
; $v0 = a & (1<<i)
; a & (1<<i) равен нулю? тогда перейти на loc_58:
        beqz    $v0, loc_58
        or      $at, $zero
; переход не случился, это значит что a & (1<<i)!=0, так что инкрементируем
"rt":
        lw      $v0, 0x18+rt($fp)
        or      $at, $zero ; load delay slot, NOP
        addiu   $v0, 1
        sw      $v0, 0x18+rt($fp)

loc_58:
; инкремент i:
        lw      $v0, 0x18+i($fp)
        or      $at, $zero ; load delay slot, NOP
        addiu   $v0, 1
        sw      $v0, 0x18+i($fp)

loc_68:
; загрузить i и сравнить его с 0x20 (32).
; перейти на loc_20, если это меньше чем 0x20 (32):
        lw      $v0, 0x18+i($fp)
        or      $at, $zero ; load delay slot, NOP
        slti   $v0, 0x20 # ' '
        bnez   $v0, loc_20
        or      $at, $zero ; branch delay slot, NOP
; эпилог функции. возврат rt:
        lw      $v0, 0x18+rt($fp)
        move    $sp, $fp ; load delay slot
        lw      $fp, 0x18+var_4($sp)
        addiu   $sp, 0x18 ; load delay slot
        jr     $ra
        or      $at, $zero ; branch delay slot, NOP

```

Это многословно: все локальные переменные расположены в локальном стеке и перезагружаются каждый раз, когда нужны. Инструкция SLLV это «Shift Word Left Logical Variable», она отличается от SLL только тем что количество бит для сдвига кодируется в SLL (и, следовательно, фиксировано), а SLL берет количество из регистра.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

## Оптимизирующий GCC

Это более сжато. Здесь две инструкции сдвигов вместо одной. Почему? Можно заменить первую инструкцию SLLV на инструкцию безусловного перехода, передав управление прямо на вторую SLLV.

Но это ещё одна инструкция перехода в функции, а от них избавляться всегда выгодно: ?? (стр. ??).

Листинг 1.302: Оптимизирующий GCC 4.4.5 (IDA)

```
f:
; $a0=a
; переменная rt будет находиться в $v0:
        move    $v0, $zero
; переменная i будет находиться в $v1:
        move    $v1, $zero
        li     $t0, 1
        li     $a3, 32
        sllv   $a1, $t0, $v1
; $a1 = $t0<<$v1 = 1<<i

loc_14:
        and    $a1, $a0
; $a1 = a&(1<<i)
; инкремент i:
        addiu  $v1, 1
; переход на loc_28 если a&(1<<i)==0 и инкремент rt:
        beqz   $a1, loc_28
        addiu  $a2, $v0, 1
; если BEQZ не сработала, сохранить обновленную rt в $v0:
        move   $v0, $a2

loc_28:
; если i!=32, перейти на loc_14 а также подготовить следующее сдвинутое
; значение:
        bne    $v1, $a3, loc_14
        sllv   $a1, $t0, $v1
; возврат
        jr     $ra
        or     $at, $zero ; branch delay slot, NOP
```

### 1.28.6. Вывод

Инструкции сдвига, аналогичные операторам Си/Си++<< и >>, в x86 это SHR/SHL (для беззнаковых значений), SAR/SHL (для знаковых значений).

Инструкции сдвига в ARM это LSR/LSL (для беззнаковых значений), ASR/LSL (для знаковых значений).

Можно также добавлять суффикс сдвига для некоторых инструкций (которые называются «data processing instructions»).

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

**Проверка определенного бита (известного на стадии компиляции)**

Проверить, присутствует ли бит 0b1000000 (0x40) в значении в регистре:

Листинг 1.303: Си/Си++

```
if (input&0x40)
    ...
```

Листинг 1.304: x86

```
TEST REG, 40h
JNZ is_set
; бит не установлен
```

Листинг 1.305: x86

```
TEST REG, 40h
JZ is_cleared
; бит установлен
```

Листинг 1.306: ARM (Режим ARM)

```
TST REG, #0x40
BNE is_set
; бит не установлен
```

Иногда AND используется вместо TEST, но флаги выставляются точно также.

**Проверка определенного бита (заданного во время исполнения)**

Это обычно происходит при помощи вот такого фрагмента на Си/Си++ (сдвинуть значение на  $n$  бит вправо, затем отрезать самый младший бит):

Листинг 1.307: Си/Си++

```
if ((value>>n)&1)
    ....
```

Это обычно реализуется в x86-коде так:

Листинг 1.308: x86

```
; REG=input_value
; CL=n
SHR REG, CL
AND REG, 1
```

Или (сдвинуть 1  $n$  раз влево, изолировать этот же бит во входном значении и проверить, не ноль ли он):

Листинг 1.309: Си/Си++

```
if (value & (1<<n))
    ....
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Это обычно так реализуется в x86-коде:

Листинг 1.310: x86

```
; CL=n  
MOV REG, 1  
SHL REG, CL  
AND input_value, REG
```

### Установка определенного бита (известного во время компиляции)

Листинг 1.311: Си/Си++

```
value=value|0x40;
```

Листинг 1.312: x86

```
OR REG, 40h
```

Листинг 1.313: ARM (Режим ARM) и ARM64

```
ORR R0, R0, #0x40
```

### Установка определенного бита (заданного во время исполнения)

Листинг 1.314: Си/Си++

```
value=value|(1<<n);
```

Это обычно так реализуется в x86-коде:

Листинг 1.315: x86

```
; CL=n  
MOV REG, 1  
SHL REG, CL  
OR input_value, REG
```

### Сброс определенного бита (известного во время компиляции)

Просто исполните операцию логического «И» (AND) с инвертированным значением:

Листинг 1.316: Си/Си++

```
value=value&(~0x40);
```

Листинг 1.317: x86

```
AND REG, 0FFFFFFBh
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

## Листинг 1.318: x64

```
AND REG, 0FFFFFFFFFFFFFFBFh
```

Это на самом деле сохранение всех бит кроме одного.

В ARM в режиме ARM есть инструкция BIC, работающая как две инструкции NOT +AND:

## Листинг 1.319: ARM (Режим ARM)

```
BIC R0, R0, #0x40
```

**Сброс определенного бита (заданного во время исполнения)**

## Листинг 1.320: Си/Си++

```
value=value&~(1<<n);
```

## Листинг 1.321: x86

```
; CL=n  
MOV REG, 1  
SHL REG, CL  
NOT REG  
AND input_value, REG
```

**1.28.7. Упражнения**

- <http://challenges.re/67>
- <http://challenges.re/68>
- <http://challenges.re/69>
- <http://challenges.re/70>

**1.29. Линейный конгруэнтный генератор как генератор псевдослучайных чисел**

Линейный конгруэнтный генератор, пожалуй, самый простой способ генерировать псевдослучайные числа.

Он не в почете в наше время<sup>154</sup>, но он настолько прост (только одно умножение, одно сложение и одна операция «И»), что мы можем использовать его в качестве примера.

---

<sup>154</sup>Вихрь Мерсенна куда лучше

```

#include <stdint.h>

// константы из книги Numerical Recipes
#define RNG_a 1664525
#define RNG_c 1013904223

static uint32_t rand_state;

void my_srand (uint32_t init)
{
    rand_state=init;
}

int my_rand ()
{
    rand_state=rand_state*RNG_a;
    rand_state=rand_state+RNG_c;
    return rand_state & 0x7fff;
}

```

Здесь две функции: одна используется для инициализации внутреннего состояния, а вторая вызывается собственно для генерации псевдослучайных чисел.

Мы видим, что в алгоритме применяются две константы. Они взяты из [William H. Press and Saul A. Teukolsky and William T. Vetterling and Brian P. Flannery, *Numerical Recipes*, (2007)]. Определим их используя выражение Си/Си++ #define. Это макрос.

Разница между макросом в Си/Си++ и константой в том, что все макросы заменяются на значения препроцессором Си/Си++ и они не занимают места в памяти как переменные.

А константы, напротив, это переменные только для чтения.

Можно взять указатель (или адрес) переменной-константы, но это невозможно сделать с макросом.

Последняя операция «И» нужна, потому что согласно стандарту Си my\_rand() должна возвращать значение в пределах 0..32767.

Если вы хотите получать 32-битные псевдослучайные значения, просто уберите последнюю операцию «И».

### 1.29.1. x86

Листинг 1.322: Оптимизирующий MSVC 2013

```

_BSS    SEGMENT
_rand_state DD  01H DUP (?)
_BSS    ENDS

_init$ = 8
_srand  PROC

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        mov     eax, DWORD PTR _init$[esp-4]
        mov     DWORD PTR _rand_state, eax
        ret     0
_srand ENDP

_TEXT  SEGMENT
_rand  PROC
        imul   eax, DWORD PTR _rand_state, 1664525
        add    eax, 1013904223 ; 3c6ef35fH
        mov     DWORD PTR _rand_state, eax
        and    eax, 32767      ; 00007fffH
        ret     0
_rand  ENDP

_TEXT  ENDS

```

Вот мы это и видим: обе константы встроены в код.

Память для них не выделяется. Функция `my_srand()` просто копирует входное значение во внутреннюю переменную `rand_state`.

`my_rand()` берет её, вычисляет следующее состояние `rand_state`, обрезает его и оставляет в регистре EAX.

Неоптимизированная версия побольше:

Листинг 1.323: Неоптимизирующий MSVC 2013

```

_BSS   SEGMENT
_rand_state DD  01H DUP (?)
_BSS   ENDS

_init$ = 8
_srand  PROC
        push   ebp
        mov    ebp, esp
        mov    eax, DWORD PTR _init$[ebp]
        mov    DWORD PTR _rand_state, eax
        pop    ebp
        ret    0
_srand  ENDP

_TEXT  SEGMENT
_rand  PROC
        push   ebp
        mov    ebp, esp
        imul  eax, DWORD PTR _rand_state, 1664525
        mov    ecx, DWORD PTR _rand_state
        add    ecx, 1013904223 ; 3c6ef35fH
        mov    DWORD PTR _rand_state, ecx
        mov    eax, DWORD PTR _rand_state
        and    eax, 32767      ; 00007fffH
        pop    ebp
        ret    0
_rand  PROC

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
_rand   ENDP
_TEXT   ENDS
```

### 1.29.2. x64

Версия для x64 почти такая же, и использует 32-битные регистры вместо 64-битных (потому что мы работаем здесь с переменными типа *int*).

Но функция `my_srand()` берет входной аргумент из регистра ECX, а не из стека:

Листинг 1.324: Оптимизирующий MSVC 2013 x64

```
_BSS    SEGMENT
rand_state DD    01H DUP (?)
_BSS    ENDS

init$ = 8
my_srand PROC
; ECX = входной аргумент
    mov    DWORD PTR rand_state, ecx
    ret    0
my_srand ENDP

_TEXT   SEGMENT
my_rand PROC
    imul   eax, DWORD PTR rand_state, 1664525 ; 0019660dH
    add    eax, 1013904223 ; 3c6ef35fH
    mov    DWORD PTR rand_state, eax
    and    eax, 32767      ; 00007fffH
    ret    0
my_rand ENDP

_TEXT   ENDS
```

GCC делает почти такой же код.

### 1.29.3. 32-bit ARM

Листинг 1.325: Оптимизирующий Keil 6/2013 (Режим ARM)

```
my_srand PROC
    LDR    r1, |L0.52| ; загрузить указатель на rand_state
    STR    r0, [r1, #0] ; сохранить rand_state
    BX    lr
    ENDP

my_rand PROC
    LDR    r0, |L0.52| ; загрузить указатель на rand_state
    LDR    r2, |L0.56| ; загрузить RNG_a
    LDR    r1, [r0, #0] ; загрузить rand_state
    MUL    r1, r2, r1
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

        LDR    r2, |L0.60| ; загрузить RNG_c
        ADD    r1, r1, r2
        STR    r1, [r0, #0] ; сохранить rand_state
; AND с 0x7FFF:
        LSL    r0, r1, #17
        LSR    r0, r0, #17
        BX    lr
        ENDP

|L0.52|
        DCD    ||.data||
|L0.56|
        DCD    0x0019660d
|L0.60|
        DCD    0x3c6ef35f

        AREA ||.data||, DATA, ALIGN=2

rand_state
        DCD    0x00000000

```

В ARM инструкцию невозможно встроить 32-битную константу, так что Keil-у приходится размещать их отдельно и дополнительно загружать. Вот еще что интересно: константу 0x7FFF также нельзя встроить. Поэтому Keil сдвигает `rand_state` влево на 17 бит и затем сдвигает вправо на 17 бит. Это аналогично Си/Си++-выражению  $(rand\_state \ll 17) \gg 17$ . Выглядит как бессмысленная операция, но тем не менее, что она делает это очищает старшие 17 бит, оставляя младшие 15 бит нетронутыми, и это наша цель, в конце концов.

Оптимизирующий Keil для режима Thumb делает почти такой же код.

#### 1.29.4. MIPS

Листинг 1.326: Оптимизирующий GCC 4.4.5 (IDA)

```

my_srand:
; записать $a0 в rand_state:
        lui    $v0, (rand_state >> 16)
        jr    $ra
        sw    $a0, rand_state

my_rand:
; загрузить rand_state в $v0:
        lui    $v1, (rand_state >> 16)
        lw    $v0, rand_state
        or    $at, $zero ; load delay slot
; умножить rand_state в $v0 на 1664525 (RNG_a):
        sll   $a1, $v0, 2
        sll   $a0, $v0, 4
        addu  $a0, $a1, $a0
        sll   $a1, $a0, 6
        subu  $a0, $a1, $a0
        addu  $a0, $a0, $v0

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        sll    $a1, $a0, 5
        addu  $a0, $a1
        sll   $a0, 3
        addu  $v0, $a0, $v0
        sll   $a0, $v0, 2
        addu  $v0, $a0
; прибавить 1013904223 (RNG_c)
; инструкция LI объединена в IDA из LUI и ORI
        li    $a0, 0x3C6EF35F
        addu  $v0, $a0
; сохранить в rand_state:
        sw    $v0, (rand_state & 0xFFFF)($v1)
        jr    $ra
        andi  $v0, 0x7FFF ; branch delay slot

```

Ух, мы видим здесь только одну константу (0x3C6EF35F или 1013904223). Где же вторая (1664525)?

Похоже, умножение на 1664525 сделано только при помощи сдвигов и прибавлений!

Проверим эту версию:

```

#define RNG_a 1664525

int f (int a)
{
    return a*RNG_a;
}

```

Листинг 1.327: Оптимизирующий GCC 4.4.5 (IDA)

```

f:
        sll    $v1, $a0, 2
        sll    $v0, $a0, 4
        addu  $v0, $v1, $v0
        sll    $v1, $v0, 6
        subu  $v0, $v1, $v0
        addu  $v0, $a0
        sll    $v1, $v0, 5
        addu  $v0, $v1
        sll    $v0, 3
        addu  $a0, $v0, $a0
        sll    $v0, $a0, 2
        jr    $ra
        addu  $v0, $a0, $v0 ; branch delay slot

```

Действительно!

### Перемещения в MIPS («relocs»)

Ещё поговорим о том, как на самом деле происходят операции загрузки из памяти и запись в память.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Листинги здесь были сделаны в IDA, которая убирает немного деталей.

Запустим objdump дважды: чтобы получить дизассемблированный листинг и список перемещений:

Листинг 1.328: Оптимизирующий GCC 4.4.5 (objdump)

```
# objdump -D rand_03.o

...

00000000 <my_srand>:
   0: 3c020000      lui    v0,0x0
   4: 03e00008      jr     ra
   8: ac440000      sw    a0,0(v0)

0000000c <my_rand>:
   c: 3c030000      lui    v1,0x0
  10: 8c620000      lw    v0,0(v1)
  14: 00200825      move  at,at
  18: 00022880      sll   a1,v0,0x2
  1c: 00022100      sll   a0,v0,0x4
  20: 00a42021      addu  a0,a1,a0
  24: 00042980      sll   a1,a0,0x6
  28: 00a42023      subu  a0,a1,a0
  2c: 00822021      addu  a0,a0,v0
  30: 00042940      sll   a1,a0,0x5
  34: 00852021      addu  a0,a0,a1
  38: 000420c0      sll   a0,a0,0x3
  3c: 00821021      addu  v0,a0,v0
  40: 00022080      sll   a0,v0,0x2
  44: 00441021      addu  v0,v0,a0
  48: 3c043c6e      lui   a0,0x3c6e
  4c: 3484f35f      ori   a0,a0,0xf35f
  50: 00441021      addu  v0,v0,a0
  54: ac620000      sw    v0,0(v1)
  58: 03e00008      jr     ra
  5c: 30427fff      andi  v0,v0,0x7fff

...

# objdump -r rand_03.o

...

RELOCATION RECORDS FOR [.text]:
OFFSET  TYPE          VALUE
00000000 R_MIPS_HI16   .bss
00000008 R_MIPS_L016   .bss
0000000c R_MIPS_HI16   .bss
00000010 R_MIPS_L016   .bss
00000054 R_MIPS_L016   .bss

...
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Рассмотрим два перемещения для функции `my_srand()`.

Первое, для адреса 0, имеет тип `R_MIPS_HI16`, и второе, для адреса 8, имеет тип `R_MIPS_L016`.

Это значит, что адрес начала сегмента `.bss` будет записан в инструкцию по адресу 0 (старшая часть адреса) и по адресу 8 (младшая часть адреса).

Ведь переменная `rand_state` находится в самом начале сегмента `.bss`.

Так что мы видим нули в операндах инструкций `LUI` и `SW` потому что там пока ничего нет — компилятор не знает, что туда записать.

Линкер это исправит и старшая часть адреса будет записана в операнд инструкции `LUI` и младшая часть адреса — в операнд инструкции `SW`.

`SW` просуммирует младшую часть адреса и то что находится в регистре `$V0` (там старшая часть).

Та же история и с функцией `my_rand()`: перемещение `R_MIPS_HI16` указывает линкеру записать старшую часть адреса сегмента `.bss` в инструкцию `LUI`.

Так что старшая часть адреса переменной `rand_state` находится в регистре `$V1`.

Инструкция `LW` по адресу `0x10` просуммирует старшую и младшую часть и загрузит значение переменной `rand_state` в `$V0`.

Инструкция `SW` по адресу `0x54` также просуммирует и затем запишет новое значение в глобальную переменную `rand_state`.

IDA обрабатывает перемещения при загрузке, и таким образом эти детали скрываются.

Но мы должны о них помнить.

### 1.29.5. Версия этого примера для многопоточной среды

Версия примера для многопоточной среды будет рассмотрена позже: ?? (стр. ??).

## 1.30. Структуры

В принципе, структура в Си/Си++ это, с некоторыми допущениями, просто всегда лежащий рядом, и в той же последовательности, набор переменных, не обязательно одного типа <sup>155</sup>.

### 1.30.1. MSVC: Пример SYSTEMTIME

Возьмем, к примеру, структуру `SYSTEMTIME`<sup>156</sup> из win32 описывающую время.

Она объявлена так:

---

<sup>155</sup> **АКА!** «гетерогенный контейнер»

<sup>156</sup> [MSDN: SYSTEMTIME structure](#)

Листинг 1.329: WinBase.h

```
typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Пишем на Си функцию для получения текущего системного времени:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME t;
    GetSystemTime (&t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
           t.wYear, t.wMonth, t.wDay,
           t.wHour, t.wMinute, t.wSecond);

    return;
};
```

Что в итоге (MSVC 2010):

Листинг 1.330: MSVC 2010 /GS-

```
_t$ = -16 ; size = 16
_main PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 16
    lea    eax, DWORD PTR _t$[ebp]
    push    eax
    call   DWORD PTR __imp__GetSystemTime@4
    movzx  ecx, WORD PTR _t$[ebp+12] ; wSecond
    push    ecx
    movzx  edx, WORD PTR _t$[ebp+10] ; wMinute
    push    edx
    movzx  eax, WORD PTR _t$[ebp+8] ; wHour
    push    eax
    movzx  ecx, WORD PTR _t$[ebp+6] ; wDay
    push    ecx
    movzx  edx, WORD PTR _t$[ebp+2] ; wMonth
    push    edx
    movzx  eax, WORD PTR _t$[ebp] ; wYear
    push    eax
    push    OFFSET $SG78811 ; '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
    call    _printf
    add     esp, 28
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main     ENDP
```

Под структуру в стеке выделено 16 байт — именно столько будет `sizeof(WORD)*8` (в структуре 8 переменных с типом `WORD`).

Обратите внимание на тот факт, что структура начинается с поля `wYear`. Можно сказать, что в качестве аргумента для `GetSystemTime()`<sup>157</sup> передается указатель на структуру `SYSTEMTIME`, но можно также сказать, что передается указатель на поле `wYear`, что одно и то же! `GetSystemTime()` пишет текущий год в тот `WORD` на который указывает переданный указатель, затем сдвигается на 2 байта вправо, пишет текущий месяц, и т. д., и т. п.

---

<sup>157</sup>[MSDN: GetSystemTime function](#)

## OllyDbg

Компилируем этот пример в MSVC 2010 с ключами /GS- /MD и запускаем в OllyDbg. Открываем окна данных и стека по адресу, который передается в качестве первого аргумента в функцию `GetSystemTime()`, ждем пока эта функция исполнится, и видим следующее:

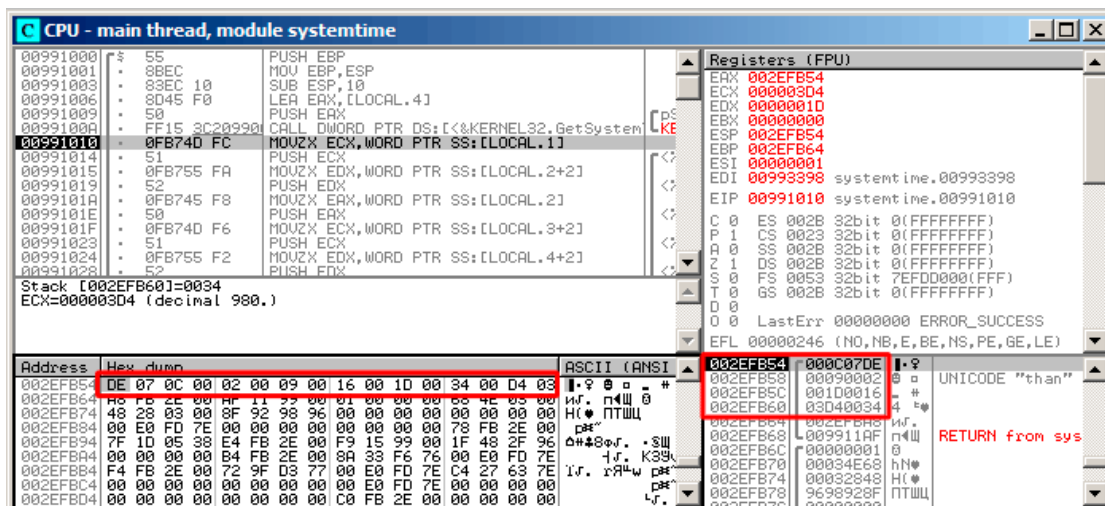


Рис. 1.104: OllyDbg: `GetSystemTime()` только что исполнилась

Точное системное время на моем компьютере, в которое исполнилась функция, это 9 декабря 2014, 22:29:52:

Листинг 1.331: Вывод `printf()`

```
2014-12-09 22:29:52
```

Таким образом, в окне данных мы видим следующие 16 байт:

```
DE 07 0C 00 02 00 09 00 16 00 1D 00 34 00 D4 03
```

Каждые два байта отражают одно поле структуры. А так как порядок байт (*endianness*) *little endian*, то в начале следует младший байт, затем старший. Следовательно, вот какие 16-битные числа сейчас записаны в памяти:

Шестнадцатеричное число	десятичное число	имя поля
0x07DE	2014	wYear
0x000C	12	wMonth
0x0002	2	wDayOfWeek
0x0009	9	wDay
0x0016	22	wHour
0x001D	29	wMinute
0x0034	52	wSecond
0x03D4	980	wMilliseconds

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

В окне стека, видны те же значения, только они сгруппированы как 32-битные значения.

Затем printf() просто берет нужные значения и выводит их на консоль.

Некоторые поля printf() не выводит (wDayOfWeek и wMilliseconds), но они находятся в памяти и доступны для использования.

### Замена структуры массивом

Тот факт, что поля структуры — это просто переменные расположенные рядом, легко проиллюстрировать следующим образом.

Глядя на описание структуры SYSTEMTIME, можно переписать этот простой пример так:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD array[8];
    GetSystemTime (array);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        array[0] /* wYear */, array[1] /* wMonth */, array[3] /* wDay */,
        array[4] /* wHour */, array[5] /* wMinute */, array[6] /* wSecond ↵
    ↵ */);

    return;
};
```

Компилятор немного ворчит:

```
systemtime2.c(7) : warning C4133: 'function' : incompatible types - from '↵
↵ WORD [8]' to 'LPSYSTEMTIME'
```

Тем не менее, выдает такой код:

#### Листинг 1.332: Неоптимизирующий MSVC 2010

```
$SG78573 DB      '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H

_array$ = -16   ; size = 16
_main PROC
    push     ebp
    mov     ebp, esp
    sub     esp, 16
    lea    eax, DWORD PTR _array$[ebp]
    push    eax
    call   DWORD PTR __imp__GetSystemTime@4
    movzx  ecx, WORD PTR _array$[ebp+12] ; wSecond
    push   ecx
    movzx  edx, WORD PTR _array$[ebp+10] ; wMinute
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

push    edx
movzx   eax, WORD PTR _array$[ebp+8] ; wHour
push    eax
movzx   ecx, WORD PTR _array$[ebp+6] ; wDay
push    ecx
movzx   edx, WORD PTR _array$[ebp+2] ; wMonth
push    edx
movzx   eax, WORD PTR _array$[ebp] ; wYear
push    eax
push    OFFSET $SG78573 ; '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
call    _printf
add     esp, 28
xor     eax, eax
mov     esp, ebp
pop     ebp
ret     0
_main  ENDP

```

И это работает так же!

Любопытно что результат на ассемблере неотличим от предыдущего. Таким образом, глядя на этот код, никогда нельзя сказать с уверенностью, была ли там объявлена структура, либо просто набор переменных.

Тем не менее, никто в здравом уме делать так не будет.

Потому что это неудобно. К тому же, иногда, поля в структуре могут меняться разработчиками, переставляться местами, и т. д.

С OllyDbg этот пример изучать не будем, потому что он будет точно такой же, как и в случае со структурой.

### 1.30.2. Выделяем место для структуры через malloc()

Однако, бывает и так, что проще хранить структуры не в стеке, а в [куче](#):

```

#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME *t;

    t=(SYSTEMTIME *)malloc (sizeof (SYSTEMTIME));

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
            t->wYear, t->wMonth, t->wDay,
            t->wHour, t->wMinute, t->wSecond);

    free (t);

    return;
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

};

Скомпилируем на этот раз с оптимизацией (/Ox) чтобы было проще увидеть то, что нам нужно.

Листинг 1.333: Оптимизирующий MSVC

```

_main      PROC
    push   esi
    push   16
    call   _malloc
    add    esp, 4
    mov    esi, eax
    push   esi
    call   DWORD PTR __imp__GetSystemTime@4
    movzx  eax, WORD PTR [esi+12] ; wSecond
    movzx  ecx, WORD PTR [esi+10] ; wMinute
    movzx  edx, WORD PTR [esi+8]  ; wHour
    push   eax
    movzx  eax, WORD PTR [esi+6]  ; wDay
    push   ecx
    movzx  ecx, WORD PTR [esi+2]  ; wMonth
    push   edx
    movzx  edx, WORD PTR [esi]    ; wYear
    push   eax
    push   ecx
    push   edx
    push   OFFSET $SG78833
    call   _printf
    push   esi
    call   _free
    add    esp, 32
    xor    eax, eax
    pop    esi
    ret    0
_main      ENDP

```

Итак, `sizeof(SYSTEMTIME) = 16`, именно столько байт выделяется при помощи `malloc()`. Она возвращает указатель на только что выделенный блок памяти в `EAX`, который копируется в `ESI`. Win32 функция `GetSystemTime()` обязуется сохранить состояние `ESI`, поэтому здесь оно нигде не сохраняется и продолжает использоваться после вызова `GetSystemTime()`.

Новая инструкция — `MOVZX` (*Move with Zero eXtend*). Она нужна почти там же где и `MOVSX`, только всегда очищает остальные биты в 0. Дело в том, что `printf()` требует 32-битный тип `int`, а в структуре лежит `WORD` — это 16-битный беззнаковый тип. Поэтому копируя значение из `WORD` в `int`, нужно очистить биты от 16 до 31, иначе там будет просто случайный мусор, оставшийся от предыдущих действий с регистрами.

В этом примере можно также представить структуру как массив 8-и `WORD`-ов:

```

#include <windows.h>
#include <stdio.h>

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

void main()
{
    WORD *t;

    t=(WORD *)malloc (16);

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t[0] /* wYear */, t[1] /* wMonth */, t[3] /* wDay */,
        t[4] /* wHour */, t[5] /* wMinute */, t[6] /* wSecond */);

    free (t);

    return;
};

```

Получим такое:

Листинг 1.334: Оптимизирующий MSVC

```

$SG78594 DB      '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H

_main PROC
    push    esi
    push    16
    call    _malloc
    add     esp, 4
    mov     esi, eax
    push    esi
    call    DWORD PTR __imp_GetSystemTime@4
    movzx  eax, WORD PTR [esi+12]
    movzx  ecx, WORD PTR [esi+10]
    movzx  edx, WORD PTR [esi+8]
    push   eax
    movzx  eax, WORD PTR [esi+6]
    push   ecx
    movzx  ecx, WORD PTR [esi+2]
    push   edx
    movzx  edx, WORD PTR [esi]
    push   eax
    push   ecx
    push   edx
    push   OFFSET $SG78594
    call   _printf
    push   esi
    call   _free
    add     esp, 32
    xor    eax, eax
    pop    esi
    ret    0
_main ENDP

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

И снова мы получаем идентичный код, неотличимый от предыдущего.

Но и снова нужно отметить, что в реальности так лучше не делать, если только вы не знаете точно, что вы делаете.

### 1.30.3. UNIX: struct tm

#### Linux

В Линуксе, для примера, возьмем структуру tm из time.h:

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    printf ("Year: %d\n", t.tm_year+1900);
    printf ("Month: %d\n", t.tm_mon);
    printf ("Day: %d\n", t.tm_mday);
    printf ("Hour: %d\n", t.tm_hour);
    printf ("Minutes: %d\n", t.tm_min);
    printf ("Seconds: %d\n", t.tm_sec);
};
```

Компилируем при помощи GCC 4.4.1:

#### Листинг 1.335: GCC 4.4.1

```
main proc near
    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFF0h
    sub     esp, 40h
    mov     dword ptr [esp], 0 ; первый аргумент для time()
    call   time
    mov     [esp+3Ch], eax
    lea    eax, [esp+3Ch] ; берем указатель на то что вернула time()
    lea    edx, [esp+10h] ; по ESP+10h будет начинаться структура struct
tm
    mov     [esp+4], edx ; передаем указатель на начало структуры
    mov     [esp], eax ; передаем указатель на результат time()
    call   localtime_r
    mov     eax, [esp+24h] ; tm_year
    lea    edx, [eax+76Ch] ; edx=eax+1900
    mov     eax, offset format ; "Year: %d\n"
    mov     [esp+4], edx
    mov     [esp], eax
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

call    printf
mov     edx, [esp+20h]      ; tm_mon
mov     eax, offset aMonthD ; "Month: %d\n"
mov     [esp+4], edx
mov     [esp], eax
call    printf
mov     edx, [esp+1Ch]     ; tm_mday
mov     eax, offset aDayD  ; "Day: %d\n"
mov     [esp+4], edx
mov     [esp], eax
call    printf
mov     edx, [esp+18h]     ; tm_hour
mov     eax, offset aHourD ; "Hour: %d\n"
mov     [esp+4], edx
mov     [esp], eax
call    printf
mov     edx, [esp+14h]     ; tm_min
mov     eax, offset aMinutesD ; "Minutes: %d\n"
mov     [esp+4], edx
mov     [esp], eax
call    printf
mov     edx, [esp+10h]
mov     eax, offset aSecondsD ; "Seconds: %d\n"
mov     [esp+4], edx      ; tm_sec
mov     [esp], eax
call    printf
leave
retn
main endp

```

К сожалению, по какой-то причине, **IDA!** не сформировала названия локальных переменных в стеке. Но так как мы уже опытные реверсеры :-)) то можем обойтись и без этого в таком простом примере.

Обратите внимание на `lea edx, [eax+76Ch]` — эта инструкция прибавляет `0x76C (1900)` к `EAX`, но не модифицирует флаги. См. также соответствующий раздел об инструкции `LEA` ([??](#) (стр. [??](#))).

## GDB

Попробуем загрузить пример в GDB [158](#):

### Листинг 1.336: GDB

```

dennis@ubuntuvms:~/polygon$ date
Mon Jun  2 18:10:37 EEST 2014
dennis@ubuntuvms:~/polygon$ gcc GCC_tm.c -o GCC_tm
dennis@ubuntuvms:~/polygon$ gdb GCC_tm
GNU gdb (GDB) 7.6.1-ubuntu
...

```

<sup>158</sup>Результат работы `date` немного подправлен в целях демонстрации. Конечно же, в реальности, нельзя так быстро запустить GDB, чтобы значение секунд осталось бы таким же.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

Reading symbols from /home/dennis/polygon/GCC_tm...(no debugging symbols ↵
↳ found)...done.
(gdb) b printf
Breakpoint 1 at 0x08048330
(gdb) run
Starting program: /home/dennis/polygon/GCC_tm

Breakpoint 1, __printf (format=0x080485c0 "Year: %d\n") at printf.c:29
29     printf.c: No such file or directory.
(gdb) x/20x $esp
0xbffff0dc:    0x080484c3      0x080485c0      0x000007de      0x00000000
0xbffff0ec:    0x08048301      0x538c93ed      0x00000025      0x0000000a
0xbffff0fc:    0x00000012      0x00000002      0x00000005      0x00000072
0xbffff10c:    0x00000001      0x00000098      0x00000001      0x00002a30
0xbffff11c:    0x0804b090      0x08048530      0x00000000      0x00000000
(gdb)

```

Мы легко находим нашу структуру в стеке. Для начала, посмотрим, как она объявлена в *time.h*:

Листинг 1.337: time.h

```

struct tm
{
    int    tm_sec;
    int    tm_min;
    int    tm_hour;
    int    tm_mday;
    int    tm_mon;
    int    tm_year;
    int    tm_wday;
    int    tm_yday;
    int    tm_isdst;
};

```

Обратите внимание что здесь 32-битные *int* вместо WORD в SYSTEMTIME. Так что, каждое поле занимает 32-битное слово.

Вот поля нашей структуры в стеке:

```

0xbffff0dc: 0x080484c3      0x080485c0      0x000007de      0x00000000
0xbffff0ec: 0x08048301      0x538c93ed      0x00000025      0x0000000a sec
0xbffff0fc: 0x00000012      hour 0x00000002      mday 0x00000005      mon 0x00000072 ↵
↳ year
0xbffff10c: 0x00000001      wday 0x00000098      yday 0x00000001      isdst 0x00002a30
0xbffff11c: 0x0804b090      0x08048530      0x00000000      0x00000000

```

Либо же, в виде таблицы:

Шестнадцатеричное число	десятичное число	имя поля
0x00000025	37	tm_sec
0x0000000a	10	tm_min
0x00000012	18	tm_hour
0x00000002	2	tm_mday
0x00000005	5	tm_mon
0x00000072	114	tm_year
0x00000001	1	tm_wday
0x00000098	152	tm_yday
0x00000001	1	tm_isdst

Как и в примере с SYSTEMTIME (?? (стр. ??)), здесь есть и другие поля, готовые для использования, но в нашем примере они не используются, например, tm\_wday, tm\_yday, tm\_isdst.

## ARM

### Оптимизирующий Keil 6/2013 (Режим Thumb)

Этот же пример:

Листинг 1.338: Оптимизирующий Keil 6/2013 (Режим Thumb)

```

var_38 = -0x38
var_34 = -0x34
var_30 = -0x30
var_2C = -0x2C
var_28 = -0x28
var_24 = -0x24
timer  = -0xC

    PUSH    {LR}
    MOVS   R0, #0           ; timer
    SUB    SP, SP, #0x34
    BL     time
    STR    R0, [SP,#0x38+timer]
    MOV    R1, SP           ; tp
    ADD    R0, SP, #0x38+timer ; timer
    BL     localtime_r
    LDR    R1, =0x76C
    LDR    R0, [SP,#0x38+var_24]
    ADDS   R1, R0, R1
    ADR    R0, aYearD       ; "Year: %d\n"
    BL     __2printf
    LDR    R1, [SP,#0x38+var_28]
    ADR    R0, aMonthD     ; "Month: %d\n"
    BL     __2printf
    LDR    R1, [SP,#0x38+var_2C]
    ADR    R0, aDayD       ; "Day: %d\n"
    BL     __2printf
    LDR    R1, [SP,#0x38+var_30]
    ADR    R0, aHourD      ; "Hour: %d\n"

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

BL    __2printf
LDR   R1, [SP,#0x38+var_34]
ADR   R0, aMinutesD ; "Minutes: %d\n"
BL    __2printf
LDR   R1, [SP,#0x38+var_38]
ADR   R0, aSecondsD ; "Seconds: %d\n"
BL    __2printf
ADD   SP, SP, #0x34
POP   {PC}

```

### Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

**IDA!** «узнала» структуру tm (потому что **IDA!** «знает» типы аргументов библиотечных функций, таких как localtime\_r()), поэтому показала здесь обращения к отдельным элементам структуры и присвоила им имена .

Листинг 1.339: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

```

var_38 = -0x38
var_34 = -0x34

PUSH {R7,LR}
MOV  R7, SP
SUB  SP, SP, #0x30
MOVS R0, #0 ; time_t *
BLX  _time
ADD  R1, SP, #0x38+var_34 ; struct tm *
STR  R0, [SP,#0x38+var_38]
MOV  R0, SP ; time_t *
BLX  _localtime_r
LDR  R1, [SP,#0x38+var_34.tm_year]
MOV  R0, 0xF44 ; "Year: %d\n"
ADD  R0, PC ; char *
ADDW R1, R1, #0x76C
BLX  _printf
LDR  R1, [SP,#0x38+var_34.tm_mon]
MOV  R0, 0xF3A ; "Month: %d\n"
ADD  R0, PC ; char *
BLX  _printf
LDR  R1, [SP,#0x38+var_34.tm_mday]
MOV  R0, 0xF35 ; "Day: %d\n"
ADD  R0, PC ; char *
BLX  _printf
LDR  R1, [SP,#0x38+var_34.tm_hour]
MOV  R0, 0xF2E ; "Hour: %d\n"
ADD  R0, PC ; char *
BLX  _printf
LDR  R1, [SP,#0x38+var_34.tm_min]
MOV  R0, 0xF28 ; "Minutes: %d\n"
ADD  R0, PC ; char *
BLX  _printf
LDR  R1, [SP,#0x38+var_34]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

MOV R0, 0xF25 ; "Seconds: %d\n"
ADD R0, PC ; char *
BLX _printf
ADD SP, SP, #0x30
POP {R7,PC}

```

...

```

00000000 tm      struct ; (sizeof=0x2C, standard type)
00000000 tm_sec  DCD ?
00000004 tm_min  DCD ?
00000008 tm_hour DCD ?
0000000C tm_mday DCD ?
00000010 tm_mon  DCD ?
00000014 tm_year DCD ?
00000018 tm_wday DCD ?
0000001C tm_yday DCD ?
00000020 tm_isdst DCD ?
00000024 tm_gmtoff DCD ?
00000028 tm_zone DCD ? ; offset
0000002C tm      ends

```

## MIPS

Листинг 1.340: Оптимизирующий GCC 4.4.5 (IDA)

```

1 main:
2
3 ; IDA не знает имен полей структуры, мы назвали их так вручную:
4
5 var_40      = -0x40
6 var_38      = -0x38
7 seconds     = -0x34
8 minutes     = -0x30
9 hour        = -0x2C
10 day         = -0x28
11 month       = -0x24
12 year        = -0x20
13 var_4       = -4
14
15 lui        $gp, (__gnu_local_gp >> 16)
16 addiu      $sp, -0x50
17 la         $gp, (__gnu_local_gp & 0xFFFF)
18 sw         $ra, 0x50+var_4($sp)
19 sw         $gp, 0x50+var_40($sp)
20 lw         $t9, (time & 0xFFFF)($gp)
21 or         $at, $zero ; load delay slot, NOP
22 jalr      $t9
23 move      $a0, $zero ; branch delay slot, NOP
24 lw         $gp, 0x50+var_40($sp)
25 addiu     $a0, $sp, 0x50+var_38
26 lw         $t9, (localtime_r & 0xFFFF)($gp)
27 addiu     $a1, $sp, 0x50+seconds

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

28 jalr $t9
29 sw $v0, 0x50+var_38($sp) ; branch delay slot
30 lw $gp, 0x50+var_40($sp)
31 lw $a1, 0x50+year($sp)
32 lw $t9, (printf & 0xFFFF)($gp)
33 la $a0, $LC0 # "Year: %d\n"
34 jalr $t9
35 addiu $a1, 1900 ; branch delay slot
36 lw $gp, 0x50+var_40($sp)
37 lw $a1, 0x50+month($sp)
38 lw $t9, (printf & 0xFFFF)($gp)
39 lui $a0, ($LC1 >> 16) # "Month: %d\n"
40 jalr $t9
41 la $a0, ($LC1 & 0xFFFF) # "Month: %d\n" ; branch delay slot
42 lw $gp, 0x50+var_40($sp)
43 lw $a1, 0x50+day($sp)
44 lw $t9, (printf & 0xFFFF)($gp)
45 lui $a0, ($LC2 >> 16) # "Day: %d\n"
46 jalr $t9
47 la $a0, ($LC2 & 0xFFFF) # "Day: %d\n" ; branch delay slot
48 lw $gp, 0x50+var_40($sp)
49 lw $a1, 0x50+hour($sp)
50 lw $t9, (printf & 0xFFFF)($gp)
51 lui $a0, ($LC3 >> 16) # "Hour: %d\n"
52 jalr $t9
53 la $a0, ($LC3 & 0xFFFF) # "Hour: %d\n" ; branch delay slot
54 lw $gp, 0x50+var_40($sp)
55 lw $a1, 0x50+minutes($sp)
56 lw $t9, (printf & 0xFFFF)($gp)
57 lui $a0, ($LC4 >> 16) # "Minutes: %d\n"
58 jalr $t9
59 la $a0, ($LC4 & 0xFFFF) # "Minutes: %d\n" ; branch delay slot
60 lw $gp, 0x50+var_40($sp)
61 lw $a1, 0x50+seconds($sp)
62 lw $t9, (printf & 0xFFFF)($gp)
63 lui $a0, ($LC5 >> 16) # "Seconds: %d\n"
64 jalr $t9
65 la $a0, ($LC5 & 0xFFFF) # "Seconds: %d\n" ; branch delay slot
66 lw $ra, 0x50+var_4($sp)
67 or $at, $zero ; load delay slot, NOP
68 jr $ra
69 addiu $sp, 0x50
70
71 $LC0: .ascii "Year: %d\n"<0>
72 $LC1: .ascii "Month: %d\n"<0>
73 $LC2: .ascii "Day: %d\n"<0>
74 $LC3: .ascii "Hour: %d\n"<0>
75 $LC4: .ascii "Minutes: %d\n"<0>
76 $LC5: .ascii "Seconds: %d\n"<0>

```

Это тот пример, где branch delay slot-ы могут нас запутать.

Например, в строке 35 есть инструкция `addiu $a1, 1900`, добавляющая 1900

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

к числу года.

Но она выполняется перед исполнением соответствующей JALR в строке 34, не забываяте.

### Структура как набор переменных

Чтобы проиллюстрировать то что структура — это просто набор переменных, лежащих в одном месте, переделаем немного пример, еще раз заглянув в описание структуры *tm*: листинг.??.

```
#include <stdio.h>
#include <time.h>

void main()
{
    int tm_sec, tm_min, tm_hour, tm_mday, tm_mon, tm_year, tm_wday, tm_yday,
    tm_isdst;
    time_t unix_time;

    unix_time=time(NULL);

    localtime_r (&unix_time, &tm_sec);

    printf ("Year: %d\n", tm_year+1900);
    printf ("Month: %d\n", tm_mon);
    printf ("Day: %d\n", tm_mday);
    printf ("Hour: %d\n", tm_hour);
    printf ("Minutes: %d\n", tm_min);
    printf ("Seconds: %d\n", tm_sec);
};
```

N.B. В *localtime\_r* передается указатель именно на *tm\_sec*, т.е. на первый элемент «структуры».

В итоге, и этот компилятор поворчит:

#### Листинг 1.341: GCC 4.7.3

```
GCC_tm2.c: In function 'main':
GCC_tm2.c:11:5: warning: passing argument 2 of 'localtime_r' from
↳ incompatible pointer type [enabled by default]
In file included from GCC_tm2.c:2:0:
/usr/include/time.h:59:12: note: expected 'struct tm *' but argument is of
↳ type 'int *'
```

Тем не менее, сгенерирует такое:

#### Листинг 1.342: GCC 4.7.3

```
main      proc near
var_30    = dword ptr -30h
var_2C    = dword ptr -2Ch
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

unix_time = dword ptr -1Ch
tm_sec    = dword ptr -18h
tm_min    = dword ptr -14h
tm_hour   = dword ptr -10h
tm_mday   = dword ptr -0Ch
tm_mon    = dword ptr -8
tm_year   = dword ptr -4

    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFFF0h
    sub     esp, 30h
    call    __main
    mov     [esp+30h+var_30], 0 ; arg 0
    call    time
    mov     [esp+30h+unix_time], eax
    lea    eax, [esp+30h+tm_sec]
    mov     [esp+30h+var_2C], eax
    lea    eax, [esp+30h+unix_time]
    mov     [esp+30h+var_30], eax
    call    localtime_r
    mov     eax, [esp+30h+tm_year]
    add     eax, 1900
    mov     [esp+30h+var_2C], eax
    mov     [esp+30h+var_30], offset aYearD ; "Year: %d\n"
    call    printf
    mov     eax, [esp+30h+tm_mon]
    mov     [esp+30h+var_2C], eax
    mov     [esp+30h+var_30], offset aMonthD ; "Month: %d\n"
    call    printf
    mov     eax, [esp+30h+tm_mday]
    mov     [esp+30h+var_2C], eax
    mov     [esp+30h+var_30], offset aDayD ; "Day: %d\n"
    call    printf
    mov     eax, [esp+30h+tm_hour]
    mov     [esp+30h+var_2C], eax
    mov     [esp+30h+var_30], offset aHourD ; "Hour: %d\n"
    call    printf
    mov     eax, [esp+30h+tm_min]
    mov     [esp+30h+var_2C], eax
    mov     [esp+30h+var_30], offset aMinutesD ; "Minutes: %d\n"
    call    printf
    mov     eax, [esp+30h+tm_sec]
    mov     [esp+30h+var_2C], eax
    mov     [esp+30h+var_30], offset aSecondsD ; "Seconds: %d\n"
    call    printf
    leave
    retn
main      endp

```

Этот код почти идентичен уже рассмотренному, и нельзя сказать, была ли структура в оригинальном исходном коде либо набор переменных.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

И это работает. Однако, в реальности так лучше не делать. Обычно, неоптимизирующий компилятор располагает переменные в локальном стеке в том же порядке, в котором они объявляются в функции.

Тем не менее, никакой гарантии нет.

Кстати, какой-нибудь другой компилятор может предупредить, что переменные `tm_year`, `tm_mon`, `tm_mday`, `tm_hour`, `tm_min`, но не `tm_sec`, используются без инициализации. Действительно, ведь компилятор не знает что они будут заполнены при вызове функции `localtime_r()`.

Мы выбрали именно этот пример для иллюстрации, потому что все члены структуры имеют тип `int`. Это не сработает, если поля структуры будут иметь размер 16 бит (`WORD`), как в случае со структурой `SYSTEMTIME` — `GetSystemTime()` заполнит их неверно (потому что локальные переменные выровнены по 32-битной границе). Читайте об этом в следующей секции: «Упаковка полей в структуре» (?? (стр. ??)).

Так что, структура — это просто набор переменных лежащих в одном месте, рядом.

Можно было бы сказать, что структура — это инструкция компилятору, заставляющая его удерживать переменные в одном месте.

Кстати, когда-то, в очень ранних версиях Си (перед 1972) структур не было вовсе [Dennis M. Ritchie, *The development of the C language*, (1993)]<sup>159</sup>.

Здесь нет примера с отладчиком: потому что он будет полностью идентичным тому, что вы уже видели.

### Структура как массив 32-битных слов

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;
    int i;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    for (i=0; i<9; i++)
    {
        int tmp=((int*)&t)[i];
        printf ("0x%08X (%d)\n", tmp, tmp);
    };
};
```

<sup>159</sup>Также доступно здесь: [pdf](#)

Мы просто приводим (*cast*) указатель на структуру к массиву *int*-ов. И это работает! Запускаем пример 23:51:45 26-July-2014.

```
0x0000002D (45)
0x00000033 (51)
0x00000017 (23)
0x0000001A (26)
0x00000006 (6)
0x00000072 (114)
0x00000006 (6)
0x000000CE (206)
0x00000001 (1)
```

Переменные здесь в том же порядке, в котором они перечислены в определении структуры: ?? (стр. ??).

Вот как это компилируется:

#### Листинг 1.343: Оптимизирующий GCC 4.8.1

```
main proc near
    push    ebp
    mov     ebp, esp
    push    esi
    push    ebx
    and     esp, 0FFFFFF0h
    sub     esp, 40h
    mov     dword ptr [esp], 0 ; timer
    lea    ebx, [esp+14h]
    call   _time
    lea    esi, [esp+38h]
    mov     [esp+4], ebx ; tp
    mov     [esp+10h], eax
    lea    eax, [esp+10h]
    mov     [esp], eax ; timer
    call   _localtime_r
    nop
    lea    esi, [esi+0] ; NOP
loc_80483D8:
; EBX здесь это указатель на структуру,
; ESI - указатель на её конец.
    mov     eax, [ebx] ; загрузить 32-битное слово из массива
    add     ebx, 4 ; следующее поле в структуре
    mov     dword ptr [esp+4], offset a0x08xD ; "0x%08X (%d)\n"
    mov     dword ptr [esp], 1
    mov     [esp+0Ch], eax ; передать значение в printf()
    mov     [esp+8], eax ; передать значение в printf()
    call   __printf_chk
    cmp     ebx, esi ; достигли конца структуры?
    jnz    short loc_80483D8 ; нет - тогда загрузить следующее значение
    lea    esp, [ebp-8]
    pop     ebx
    pop     esi
    pop     ebp
    retn
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
main endp
```

И действительно: место в локальном стеке в начале используется как структура, затем как массив.

Возможно даже модифицировать поля структуры через указатель.

И снова, это сомнительный хакерский способ, который не рекомендуется использовать в настоящем коде.

### Упражнение

В качестве упражнения, попробуйте модифицировать (увеличить на 1) текущий номер месяца обращаясь со структурой как с массивом.

### Структура как массив байт

Можно пойти еще дальше. Можно привести (*cast*) указатель к массиву байт и вывести его:

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;
    int i, j;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    for (i=0; i<9; i++)
    {
        for (j=0; j<4; j++)
            printf ("0x%02X ", ((unsigned char*)&t)[i*4+j]);
        printf ("\n");
    }
};
```

```
0x2D 0x00 0x00 0x00
0x33 0x00 0x00 0x00
0x17 0x00 0x00 0x00
0x1A 0x00 0x00 0x00
0x06 0x00 0x00 0x00
0x72 0x00 0x00 0x00
0x06 0x00 0x00 0x00
0xCE 0x00 0x00 0x00
0x01 0x00 0x00 0x00
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Мы также запускаем этот пример в 23:51:45 26-July-2014<sup>160</sup>. Переменные точно такие же, как и в предыдущем выводе (?? (стр. ??)), и конечно, младший байт идет в самом начале, потому что это архитектура little-endian (?? (стр. ??)).

Листинг 1.344: Оптимизирующий GCC 4.8.1

```
main proc near
    push    ebp
    mov     ebp, esp
    push    edi
    push    esi
    push    ebx
    and     esp, 0FFFFFF0h
    sub     esp, 40h
    mov     dword ptr [esp], 0 ; timer
    lea     esi, [esp+14h]
    call    _time
    lea     edi, [esp+38h] ; struct end
    mov     [esp+4], esi ; tp
    mov     [esp+10h], eax
    lea     eax, [esp+10h]
    mov     [esp], eax ; timer
    call    _localtime_r
    lea     esi, [esi+0] ; NOP
; ESI здесь это указатель на структуру в локальном стеке.
; EDI это указатель на конец структуры.
loc_8048408:
    xor     ebx, ebx ; j=0

loc_804840A:
    movzx   eax, byte ptr [esi+ebx] ; загрузить байт
    add     ebx, 1 ; j=j+1
    mov     dword ptr [esp+4], offset a0x02x ; "0x%02X "
    mov     dword ptr [esp], 1
    mov     [esp+8], eax ; передать загруженный байт в printf()
    call    ___printf_chk
    cmp     ebx, 4
    jnz     short loc_804840A
; вывести символ перевода каретки (CR)
    mov     dword ptr [esp], 0Ah ; c
    add     esi, 4
    call    _putchar
    cmp     esi, edi ; достигли конца структуры?
    jnz     short loc_8048408 ; j=0
    lea     esp, [ebp-0Ch]
    pop     ebx
    pop     esi
    pop     edi
    pop     ebp
    retn
main endp
```

<sup>160</sup>Время и дата такая же в целях демонстрации. Значения байт были подправлены.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



## GNU Scientific Library: Представление комплексных чисел

Это относительно редкий случай, когда массив используется вместо структуры, намеренно:

### Representation of complex numbers

=====

Complex numbers are represented using the type `gsl_complex`. The internal representation of this type may vary across platforms and should not be accessed directly. The functions and macros described below allow complex numbers to be manipulated in a portable way.

For reference, the default form of the `gsl_complex` type is given by the following struct:

```
typedef struct
{
    double dat[2];
} gsl_complex;
```

The real and imaginary part are stored in contiguous elements of a two element array. This eliminates any padding between the real and imaginary parts, `dat[0]` and `dat[1]`, allowing the struct to be mapped correctly onto packed complex arrays.

( [URL](#) )

### 1.30.4. Упаковка полей в структуре

Достаточно немаловажный момент, это упаковка полей в структурах.

Возьмем простой пример:

```
#include <stdio.h>

struct s
{
    char a;
    int b;
    char c;
    int d;
};

void f(struct s s)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", s.a, s.b, s.c, s.d);
};

int main()
{
    struct s tmp;
    tmp.a=1;
    tmp.b=2;
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

tmp.c=3;
tmp.d=4;
f(tmp);
};

```

Как видно, мы имеем два поля *char* (занимающий один байт) и еще два — *int* (по 4 байта).

## x86

Компилируется это все в:

Листинг 1.345: MSVC 2012 /GS- /Ob0

```

1  _tmp$ = -16
2  _main PROC
3      push    ebp
4      mov     ebp, esp
5      sub     esp, 16
6      mov     BYTE PTR _tmp$[ebp], 1      ; установить поле a
7      mov     DWORD PTR _tmp$[ebp+4], 2  ; установить поле b
8      mov     BYTE PTR _tmp$[ebp+8], 3   ; установить поле c
9      mov     DWORD PTR _tmp$[ebp+12], 4 ; установить поле d
10     sub     esp, 16                    ; выделить место для временной
      структуры
11     mov     eax, esp
12     mov     ecx, DWORD PTR _tmp$[ebp]  ; скопировать нашу структуру во
      временную
13     mov     DWORD PTR [eax], ecx
14     mov     edx, DWORD PTR _tmp$[ebp+4]
15     mov     DWORD PTR [eax+4], edx
16     mov     ecx, DWORD PTR _tmp$[ebp+8]
17     mov     DWORD PTR [eax+8], ecx
18     mov     edx, DWORD PTR _tmp$[ebp+12]
19     mov     DWORD PTR [eax+12], edx
20     call    _f
21     add     esp, 16
22     xor     eax, eax
23     mov     esp, ebp
24     pop     ebp
25     ret     0
26 _main ENDP
27
28 _s$ = 8 ; size = 16
29 ?f@@YAXUs@@@Z PROC ; f
30     push    ebp
31     mov     ebp, esp
32     mov     eax, DWORD PTR _s$[ebp+12]
33     push    eax
34     movsx   ecx, BYTE PTR _s$[ebp+8]
35     push    ecx
36     mov     edx, DWORD PTR _s$[ebp+4]
37     push    edx
38     movsx   eax, BYTE PTR _s$[ebp]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

39     push    eax
40     push    OFFSET $SG3842
41     call   _printf
42     add    esp, 20
43     pop    ebp
44     ret    0
45     ?f@@YAXUs@@@Z ENDP ; f
46     _TEXT    ENDS

```

Кстати, мы передаем всю структуру, но в реальности, как видно, структура в начале копируется во временную структуру (выделение места под нее в стеке происходит в строке 10, а все 4 поля, по одному, копируются в строках 12 ... 19), затем передается только указатель на нее (или адрес).

Структура копируется, потому что неизвестно, будет ли функция `f()` модифицировать структуру или нет. И если да, то структура внутри `main()` должна остаться той же.

Мы могли бы использовать указатели на Си/Си++, и итоговый код был бы почти такой же, только копирования не было бы.

Мы видим здесь что адрес каждого поля в структуре выравнивается по 4-байтной границе. Так что каждый `char` здесь занимает те же 4 байта что и `int`. Зачем? Затем что процессору удобнее обращаться по таким адресам и кэшировать данные из памяти.

Но это не экономично по размеру данных.

Попробуем скомпилировать тот же исходник с опцией `(/Zp1) (/Zp[n] pack structures on n-byte boundary)`.

Листинг 1.346: MSVC 2012 /GS- /Zp1

```

1  _main    PROC
2      push    ebp
3      mov     ebp, esp
4      sub     esp, 12
5      mov     BYTE PTR _tmp$[ebp], 1      ; установить поле a
6      mov     DWORD PTR _tmp$[ebp+1], 2  ; установить поле b
7      mov     BYTE PTR _tmp$[ebp+5], 3   ; установить поле c
8      mov     DWORD PTR _tmp$[ebp+6], 4  ; установить поле d
9      sub     esp, 12                    ; выделить место для временной
    структуры
10     mov     eax, esp
11     mov     ecx, DWORD PTR _tmp$[ebp]   ; скопировать 10 байт
12     mov     DWORD PTR [eax], ecx
13     mov     edx, DWORD PTR _tmp$[ebp+4]
14     mov     DWORD PTR [eax+4], edx
15     mov     cx, WORD PTR _tmp$[ebp+8]
16     mov     WORD PTR [eax+8], cx
17     call   _f
18     add     esp, 12
19     xor     eax, eax
20     mov     esp, ebp
21     pop     ebp

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

22     ret     0
23 _main     ENDP
24
25 _TEXT     SEGMENT
26 _s$ = 8 ; size = 10
27 ?f@@YAXUs@@@Z PROC     ; f
28     push   ebp
29     mov    ebp, esp
30     mov    eax, DWORD PTR _s$[ebp+6]
31     push   eax
32     movsx  ecx, BYTE PTR _s$[ebp+5]
33     push   ecx
34     mov    edx, DWORD PTR _s$[ebp+1]
35     push   edx
36     movsx  eax, BYTE PTR _s$[ebp]
37     push   eax
38     push   OFFSET $SG3842
39     call  _printf
40     add    esp, 20
41     pop    ebp
42     ret    0
43 ?f@@YAXUs@@@Z ENDP     ; f

```

Теперь структура занимает 10 байт и все *char* занимают по байту. Что это дает? Экономии места. Недостаток — процессор будет обращаться к этим полям не так эффективно по скорости, как мог бы.

Структура так же копируется в `main()`. Но не по одному полю, а 10 байт, при помощи трех пар `MOV`.

Почему не 4? Компилятор рассудил, что будет лучше скопировать 10 байт при помощи 3 пар `MOV`, чем копировать два 32-битных слова и два байта при помощи 4 пар `MOV`.

Кстати, подобная реализация копирования при помощи `MOV` взамен вызова функции `memcpy()`, например, это очень распространенная практика, потому что это в любом случае работает быстрее чем вызов `memcpy()` — если речь идет о коротких блоках, конечно: ?? (стр. ??).

Как нетрудно догадаться, если структура используется много в каких исходниках и объектных файлах, все они должны быть откомпилированы с одним и тем же соглашением об упаковке структур.

Помимо ключа `MSVC /Zp`, указывающего, по какой границе упаковывать поля структур, есть также опция компилятора `#pragma pack`, её можно указывать прямо в исходнике. Это справедливо и для `MSVC`<sup>161</sup> и `GCC`<sup>162</sup>.

Давайте теперь вернемся к `SYSTEMTIME`, которая состоит из 16-битных полей. Откуда наш компилятор знает что их надо паковать по однобайтной границе?

В файле `WinNT.h` попадаете такое:

<sup>161</sup>[MSDN: Working with Packing Structures](#)

<sup>162</sup>[Structure-Packing Pragmas](#)

## Листинг 1.347: WinNT.h

```
#include "pshpack1.h"
```

И такое:

## Листинг 1.348: WinNT.h

```
#include "pshpack4.h" // 4 byte packing is the default
```

Сам файл PshPack1.h выглядит так:

## Листинг 1.349: PshPack1.h

```
#if ! (defined(lint) || defined(RC_INVOKED))
#if ( _MSC_VER >= 800 && !defined(_M_I86)) || defined(_PUSHPOP_SUPPORTED)
#pragma warning(disable:4103)
#if !(defined( MIDL_PASS )) || defined( __midl )
#pragma pack(push,1)
#else
#pragma pack(1)
#endif
#else
#pragma pack(1)
#endif
#endif /* ! (defined(lint) || defined(RC_INVOKED)) */
```

Собственно, так и задается компилятору, как паковать объявленные после `#pragma pack` структуры.

## OllyDbg + упаковка полей по умолчанию

Попробуем в OllyDbg наш пример, где поля выровнены по умолчанию (4 байта):

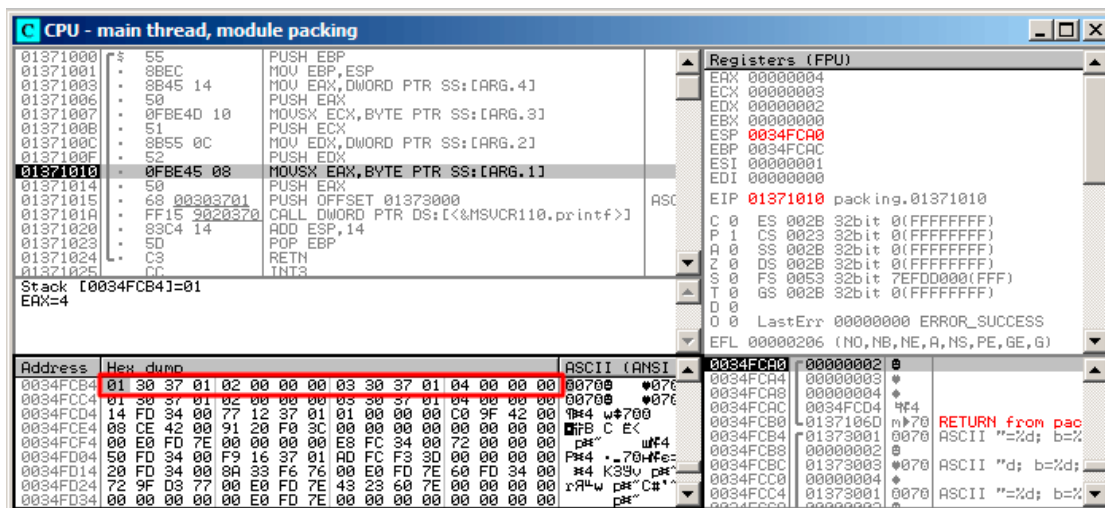


Рис. 1.105: OllyDbg: Перед исполнением printf()

В окне данных видим наши четыре поля. Вот только, откуда взялись случайные байты (0x30, 0x37, 0x01) рядом с первым (а) и третьим (с) полем?

Если вернетесь к листингу ?? (стр. ??), то увидите, что первое и третье поле имеет тип *char*, а следовательно, туда записывается только один байт, 1 и 3 соответственно (строки 6 и 8).

Остальные три байта 32-битного слова не будут модифицироваться в памяти!

А, следовательно, там остается случайный мусор. Этот мусор никак не будет влиять на работу printf(), потому что значения для нее готовятся при помощи инструкции MOVSB, которая загружает из памяти байты а не слова: листинг ?? (строки 34 и 38).

Кстати, здесь используется именно MOVSB (расширяющая знак), потому что тип *char*— знаковый по умолчанию в MSVC и GCC.

Если бы здесь был тип unsigned char или uint8\_t, то здесь была бы инструкция MOVZX.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

## OllyDbg + упаковка полей по границе в 1 байт

Здесь всё куда понятнее: 4 поля занимают 10 байт и значения сложены в памяти друг к другу

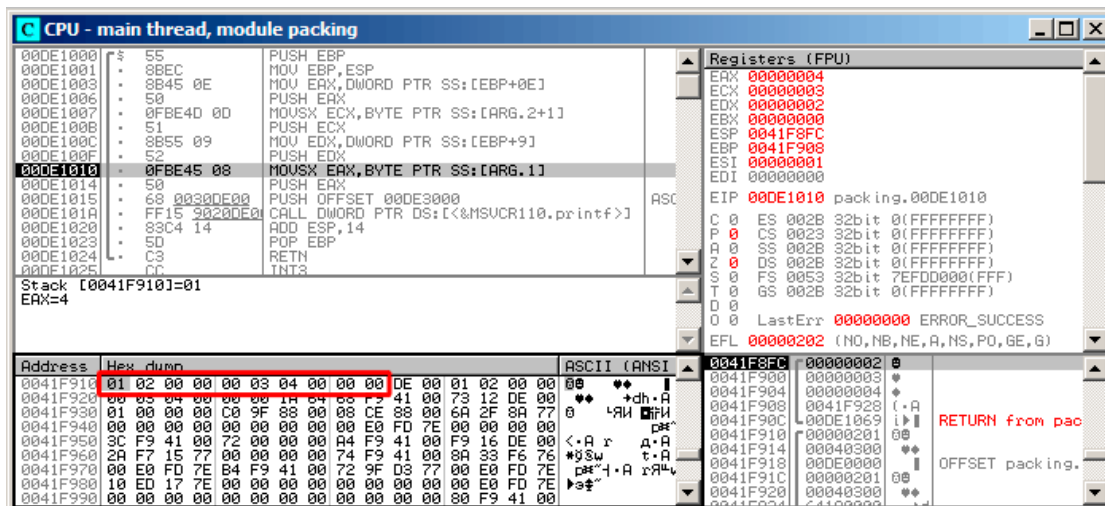


Рис. 1.106: OllyDbg: Перед исполнением printf()

## ARM

### Оптимизирующий Keil 6/2013 (Режим Thumb)

Листинг 1.350: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
.text:0000003E          exit ; CODE XREF: f+16
.text:0000003E 05 B0          ADD     SP, SP, #0x14
.text:00000040 00 BD          POP     {PC}

.text:00000280          f
.text:00000280
.text:00000280          var_18 = -0x18
.text:00000280          a      = -0x14
.text:00000280          b      = -0x10
.text:00000280          c      = -0xC
.text:00000280          d      = -8
.text:00000280 0F B5          PUSH   {R0-R3,LR}
.text:00000282 81 B0          SUB     SP, SP, #4
.text:00000284 04 98          LDR    R0, [SP,#16] ; d
.text:00000286 02 9A          LDR    R2, [SP,#8] ; b
.text:00000288 00 90          STR    R0, [SP]
.text:0000028A 68 46          MOV    R0, SP
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

.text:0000028C 03 7B	LDRB	R3, [R0,#12]	; c
.text:0000028E 01 79	LDRB	R1, [R0,#4]	; a
.text:00000290 59 A0	ADR	R0, aADBDCDDD	; "a=%d; b=%d; c=%d; d=%d\n"
.text:00000292 05 F0 AD FF	BL	__2printf	
.text:00000296 D2 E6	B	exit	

Как мы помним, здесь передается не указатель на структуру, а сама структура, а так как в ARM первые 4 аргумента функции передаются через регистры, то поля структуры передаются через R0-R3.

Инструкция LDRB загружает один байт из памяти и расширяет до 32-бит учитывая знак.

Это то же что и инструкция MOVSB в x86. Она здесь применяется для загрузки полей *a* и *c* из структуры.

Еще что бросается в глаза, так это то что вместо эпилога функции, переход на эпилог другой функции!

Действительно, то была совсем другая, не относящаяся к этой, функция, однако, она имела точно такой же эпилог (видимо, тоже хранила в стеке 5 локальных переменных ( $5 * 4 = 0x14$ )). К тому же, она находится рядом (обратите внимание на адреса).

Действительно, нет никакой разницы, какой эпилог исполнять, если он работает так же, как нам нужно.

Keil решил использовать часть другой функции, вероятно, из-за экономии.

Эпилог занимает 4 байта, а переход — только 2.

### ARM + Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

Листинг 1.351: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

```
var_C = -0xC

    PUSH   {R7,LR}
    MOV    R7, SP
    SUB    SP, SP, #4
    MOV    R9, R1 ; b
    MOV    R1, R0 ; a
    MOVW   R0, #0xF10 ; "a=%d; b=%d; c=%d; d=%d\n"
    SXTB   R1, R1 ; подготовить a
    MOVT.W R0, #0
    STR    R3, [SP,#0xC+var_C] ; сохранить d в стек для printf()
    ADD    R0, PC ; строка формата
    SXTB   R3, R2 ; подготовить c
    MOV    R2, R9 ; b
    BLX   __printf
    ADD    SP, SP, #4
    POP    {R7,PC}
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



SXTB (*Signed Extend Byte*) это также аналог MOVSBX в x86. Всё остальное — так же.

## MIPS

Листинг 1.352: Оптимизирующий GCC 4.4.5 (IDA)

```

1 f:
2
3 var_18      = -0x18
4 var_10      = -0x10
5 var_4       = -4
6 arg_0       = 0
7 arg_4       = 4
8 arg_8       = 8
9 arg_C       = 0xC
10
11 ; $a0=s.a
12 ; $a1=s.b
13 ; $a2=s.c
14 ; $a3=s.d
15         lui    $gp, (__gnu_local_gp >> 16)
16         addiu  $sp, -0x28
17         la     $gp, (__gnu_local_gp & 0xFFFF)
18         sw     $ra, 0x28+var_4($sp)
19         sw     $gp, 0x28+var_10($sp)
20 ; подготовить байт из 32-битного big-endian значения:
21         sra    $t0, $a0, 24
22         move   $v1, $a1
23 ; подготовить байт из 32-битного big-endian значения:
24         sra    $v0, $a2, 24
25         lw     $t9, (printf & 0xFFFF)($gp)
26         sw     $a0, 0x28+arg_0($sp)
27         lui    $a0, ($LC0 >> 16) # "a=%d; b=%d; c=%d; d=%d\n"
28         sw     $a3, 0x28+var_18($sp)
29         sw     $a1, 0x28+arg_4($sp)
30         sw     $a2, 0x28+arg_8($sp)
31         sw     $a3, 0x28+arg_C($sp)
32         la     $a0, ($LC0 & 0xFFFF) # "a=%d; b=%d; c=%d; d=%d\n"
33         move   $a1, $t0
34         move   $a2, $v1
35         jalr   $t9
36         move   $a3, $v0 ; branch delay slot
37         lw     $ra, 0x28+var_4($sp)
38         or     $at, $zero ; load delay slot, NOP
39         jr     $ra
40         addiu  $sp, 0x28 ; branch delay slot
41
42 $LC0:      .ascii "a=%d; b=%d; c=%d; d=%d\n"<0>

```

Поля структуры приходят в регистрах \$A0..\$A3 и затем перетасовываются в регистры \$A1..\$A3 для printf(), в то время как 4-е поле (из \$A3) передается через локальный стек используя SW.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Но здесь есть две инструкции SRA («Shift Word Right Arithmetic»), которые готовят поля типа *char*.

Почему? По умолчанию, MIPS это big-endian архитектура ?? (стр. ??), и Debian Linux в котором мы работаем, также big-endian.

Так что когда один байт расположен в 32-битном элементе структуры, он занимает биты 31..24.

И когда переменную типа *char* нужно расширить до 32-битного значения, она должна быть сдвинута вправо на 24 бита.

*char* это знаковый тип, так что здесь нужно использовать арифметический сдвиг вместо логического.

### Еще кое-что

Передача структуры как аргумент функции (вместо передачи указателя на структуру) это то же что и передача всех полей структуры по одному.

Если поля в структуре пакуются по умолчанию, то функцию *f()* можно переписать так:

```
void f(char a, int b, char c, int d)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", a, b, c, d);
};
```

И в итоге будет такой же код.

### 1.30.5. Вложенные структуры

Теперь, как насчет ситуаций, когда одна структура определена внутри другой структуры?

```
#include <stdio.h>

struct inner_struct
{
    int a;
    int b;
};

struct outer_struct
{
    char a;
    int b;
    struct inner_struct c;
    char d;
    int e;
};

void f(struct outer_struct s)
{
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    printf ("a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d\n",
           s.a, s.b, s.c.a, s.c.b, s.d, s.e);
};

int main()
{
    struct outer_struct s;
    s.a=1;
    s.b=2;
    s.c.a=100;
    s.c.b=101;
    s.d=3;
    s.e=4;
    f(s);
};

```

...в этом случае, оба поля `inner_struct` просто будут располагаться между полями `a,b` и `d,e` в `outer_struct`.

Компилируем (MSVC 2010):

Листинг 1.353: Оптимизирующий MSVC 2010 /Ob0

```

$SG2802 DB    'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d', 0aH, 00H

_TEXT      SEGMENT
_s$ = 8
_f        PROC
    mov     eax, DWORD PTR _s$[esp+16]
    movsx  ecx, BYTE PTR _s$[esp+12]
    mov     edx, DWORD PTR _s$[esp+8]
    push   eax
    mov     eax, DWORD PTR _s$[esp+8]
    push   ecx
    mov     ecx, DWORD PTR _s$[esp+8]
    push   edx
    movsx  edx, BYTE PTR _s$[esp+8]
    push   eax
    push   ecx
    push   edx
    push   OFFSET $SG2802 ; 'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d'
    call   _printf
    add     esp, 28
    ret    0
_f        ENDP

_s$ = -24
_main     PROC
    sub     esp, 24
    push   ebx
    push   esi
    push   edi
    mov     ecx, 2
    sub     esp, 24

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov    eax, esp
; С этого момента, EAX это синоним ESP:
mov    BYTE PTR _s$(esp+60), 1
mov    ebx, DWORD PTR _s$(esp+60)
mov    DWORD PTR [eax], ebx
mov    DWORD PTR [eax+4], ecx
lea    edx, DWORD PTR [ecx+98]
lea    esi, DWORD PTR [ecx+99]
lea    edi, DWORD PTR [ecx+2]
mov    DWORD PTR [eax+8], edx
mov    BYTE PTR _s$(esp+76), 3
mov    ecx, DWORD PTR _s$(esp+76)
mov    DWORD PTR [eax+12], esi
mov    DWORD PTR [eax+16], ecx
mov    DWORD PTR [eax+20], edi
call   _f
add    esp, 24
pop    edi
pop    esi
xor    eax, eax
pop    ebx
add    esp, 24
ret    0
_main  ENDP

```

Очень любопытный момент в том, что глядя на этот код на ассемблере, мы даже не видим, что была использована какая-то еще другая структура внутри этой! Так что, пожалуй, можно сказать, что все вложенные структуры в итоге разворачиваются в одну, *линейную* или *одномерную* структуру.

Конечно, если заменить объявление `struct inner_struct c`; на `struct inner_struct *c`; (объявляя таким образом указатель), ситуация будет совсем иная.

## OllyDbg

Загружаем пример в OllyDbg и смотрим на `outer_struct` в памяти:

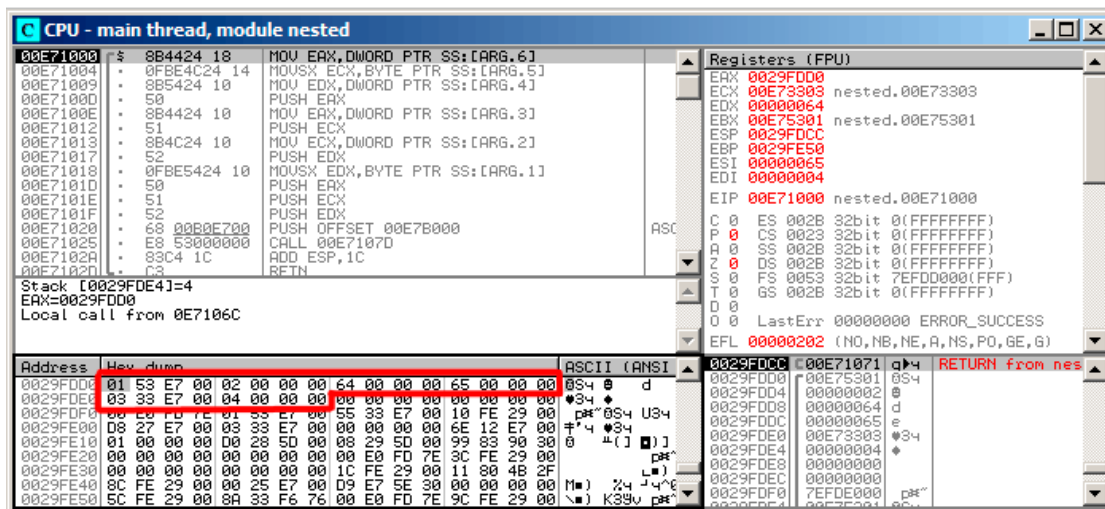


Рис. 1.107: OllyDbg: Перед исполнением `printf()`

Вот как расположены значения в памяти:

- (`outer_struct.a`) (байт) 1 + 3 байта случайного мусора;
- (`outer_struct.b`) (32-битное слово) 2;
- (`inner_struct.a`) (32-битное слово) 0x64 (100);
- (`inner_struct.b`) (32-битное слово) 0x65 (101);
- (`outer_struct.d`) (байт) 3 + 3 байта случайного мусора;
- (`outer_struct.e`) (32-битное слово) 4.

### 1.30.6. Работа с битовыми полями в структуре

#### Пример CPUID

Язык Си/Си++ позволяет указывать, сколько именно бит отвести для каждого поля структуры. Это удобно если нужно экономить место в памяти. К примеру, для переменной типа `bool` достаточно одного бита. Но, это не очень удобно, если нужна скорость.

Рассмотрим пример с инструкцией `CPUID`<sup>163</sup>. Эта инструкция возвращает информацию о том, какой процессор имеется в наличии и какие возможности он имеет.

<sup>163</sup>[wikipedia](#)

Если перед исполнением инструкции в EAX будет 1, то CPUID вернет упакованную в EAX такую информацию о процессоре:

3:0 (4 бита)	Stepping
7:4 (4 бита)	Model
11:8 (4 бита)	Family
13:12 (2 бита)	Processor Type
19:16 (4 бита)	Extended Model
27:20 (8 бит)	Extended Family

MSVC 2010 имеет макрос для CPUID, а GCC 4.4.1 — нет. Поэтому для GCC сделаем эту функцию сами, используя его встроенный ассемблер<sup>164</sup>.

```
#include <stdio.h>

#ifdef __GNUC__
static inline void cpuid(int code, int *a, int *b, int *c, int *d) {
    asm volatile("cpuid":"=a"(*a), "=b"(*b), "=c"(*c), "=d"(*d):"a"(code));
}
#endif

#ifdef _MSC_VER
#include <intrin.h>
#endif

struct CPUID_1_EAX
{
    unsigned int stepping:4;
    unsigned int model:4;
    unsigned int family_id:4;
    unsigned int processor_type:2;
    unsigned int reserved1:2;
    unsigned int extended_model_id:4;
    unsigned int extended_family_id:8;
    unsigned int reserved2:4;
};

int main()
{
    struct CPUID_1_EAX *tmp;
    int b[4];

#ifdef _MSC_VER
    __cpuid(b,1);
#endif

#ifdef __GNUC__
    cpuid (1, &b[0], &b[1], &b[2], &b[3]);
#endif

    tmp=(struct CPUID_1_EAX *)&b[0];
```

<sup>164</sup> [Подробнее о встроенном ассемблере GCC](#)

```

printf ("stepping=%d\n", tmp->stepping);
printf ("model=%d\n", tmp->model);
printf ("family_id=%d\n", tmp->family_id);
printf ("processor_type=%d\n", tmp->processor_type);
printf ("extended_model_id=%d\n", tmp->extended_model_id);
printf ("extended_family_id=%d\n", tmp->extended_family_id);

return 0;
};

```

После того как CPUID заполнит EAX/EBX/ECX/EDX, у нас они отразятся в массиве `b[]`. Затем, мы имеем указатель на структуру `CPUID_1_EAX`, и мы указываем его на значение EAX из массива `b[]`.

Иными словами, мы трактуем 32-битный *int* как структуру.

Затем мы читаем отдельные биты из структуры.

## MSVC

Компилируем в MSVC 2008 с опцией `/Ox`:

Листинг 1.354: Оптимизирующий MSVC 2008

```

_b$ = -16 ; size = 16
_main PROC
    sub     esp, 16
    push   ebx

    xor    ecx, ecx
    mov    eax, 1
    cpuid
    push   esi
    lea   esi, DWORD PTR _b$[esp+24]
    mov   DWORD PTR [esi], eax
    mov   DWORD PTR [esi+4], ebx
    mov   DWORD PTR [esi+8], ecx
    mov   DWORD PTR [esi+12], edx

    mov   esi, DWORD PTR _b$[esp+24]
    mov   eax, esi
    and   eax, 15
    push  eax
    push  OFFSET $SG15435 ; 'stepping=%d', 0aH, 00H
    call  _printf

    mov   ecx, esi
    shr   ecx, 4
    and   ecx, 15
    push  ecx
    push  OFFSET $SG15436 ; 'model=%d', 0aH, 00H
    call  _printf

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
mov     edx, esi
shr     edx, 8
and     edx, 15
push    edx
push    OFFSET $SG15437 ; 'family_id=%d', 0aH, 00H
call    _printf

mov     eax, esi
shr     eax, 12
and     eax, 3
push    eax
push    OFFSET $SG15438 ; 'processor_type=%d', 0aH, 00H
call    _printf

mov     ecx, esi
shr     ecx, 16
and     ecx, 15
push    ecx
push    OFFSET $SG15439 ; 'extended_model_id=%d', 0aH, 00H
call    _printf

shr     esi, 20
and     esi, 255
push    esi
push    OFFSET $SG15440 ; 'extended_family_id=%d', 0aH, 00H
call    _printf
add     esp, 48
pop     esi

xor     eax, eax
pop     ebx

add     esp, 16
ret     0
_main   ENDP
```

Инструкция SHR сдвигает значение из EAX на то количество бит, которое нужно *пропустить*, то есть, мы игнорируем некоторые биты *справа*.

А инструкция AND очищает биты *слева* которые нам не нужны, или же, говоря иначе, она оставляет по маске только те биты в EAX, которые нам сейчас нужны.



## MSVC + OllyDbg

Загрузим пример в OllyDbg и увидим, какие значения были установлены в EAX/EBX/ECX/EDX после исполнения CPUID:

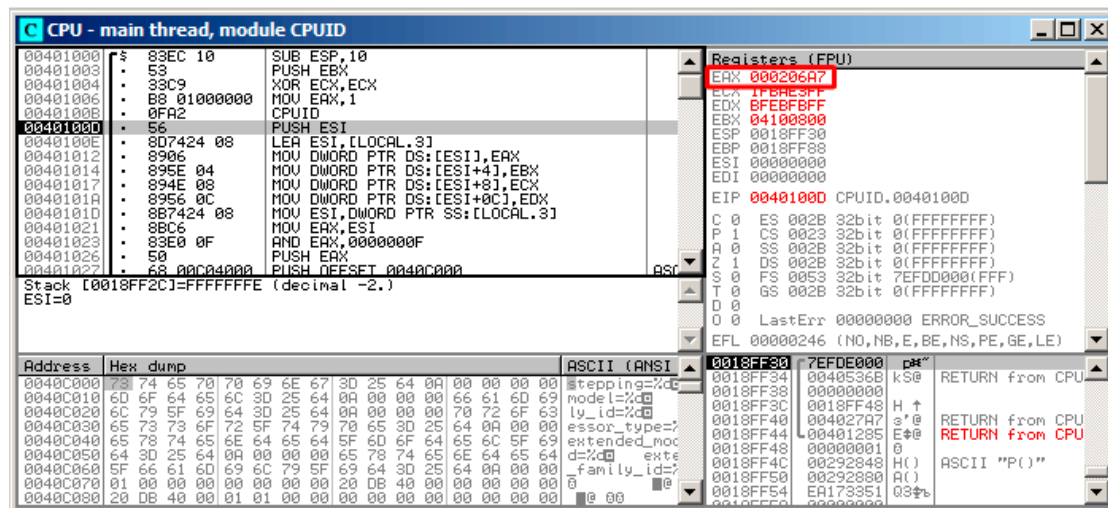


Рис. 1.108: OllyDbg: После исполнения CPUID

В EAX установлено 0x000206A7 (мой **CPU!** — Intel Xeon E3-1220).

В двоичном виде это 0b00000000000000100000011010100111.

Вот как распределяются биты по полям в моем случае:

поле	в двоичном виде	в десятичном виде
reserved2	0000	0
extended_family_id	00000000	0
extended_model_id	0010	2
reserved1	00	0
processor_id	00	0
family_id	0110	6
model	1010	10
stepping	0111	7

Листинг 1.355: Вывод в консоль

```
stepping=7
model=10
family_id=6
processor_type=0
extended_model_id=2
extended_family_id=0
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

**GCC**

Попробуем GCC 4.4.1 с опцией -O3.

Листинг 1.356: Оптимизирующий GCC 4.4.1

```

main          proc near ; DATA XREF: _start+17
  push        ebp
  mov         ebp, esp
  and         esp, 0FFFFFF0h
  push        esi
  mov         esi, 1
  push        ebx
  mov         eax, esi
  sub         esp, 18h
  cpuid
  mov         esi, eax
  and         eax, 0Fh
  mov         [esp+8], eax
  mov         dword ptr [esp+4], offset aSteppingD ; "stepping=%d\n"
  mov         dword ptr [esp], 1
  call        __printf_chk
  mov         eax, esi
  shr         eax, 4
  and         eax, 0Fh
  mov         [esp+8], eax
  mov         dword ptr [esp+4], offset aModelD ; "model=%d\n"
  mov         dword ptr [esp], 1
  call        __printf_chk
  mov         eax, esi
  shr         eax, 8
  and         eax, 0Fh
  mov         [esp+8], eax
  mov         dword ptr [esp+4], offset aFamily_idD ; "family_id=%d\n"
  mov         dword ptr [esp], 1
  call        __printf_chk
  mov         eax, esi
  shr         eax, 0Ch
  and         eax, 3
  mov         [esp+8], eax
  mov         dword ptr [esp+4], offset aProcessor_type ; "processor_type=%d\n"
  mov         dword ptr [esp], 1
  call        __printf_chk
  mov         eax, esi
  shr         eax, 10h
  shr         esi, 14h
  and         eax, 0Fh
  and         esi, 0FFh
  mov         [esp+8], eax
  mov         dword ptr [esp+4], offset aExtended_model ;
"extended_model_id=%d\n"
  mov         dword ptr [esp], 1
  call        __printf_chk
  mov         [esp+8], esi

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

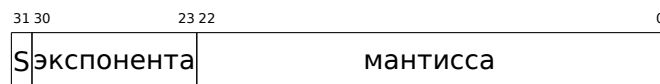
mov     dword ptr [esp+4], offset unk_80486D0
mov     dword ptr [esp], 1
call   __printf_chk
add     esp, 18h
xor     eax, eax
pop     ebx
pop     esi
mov     esp, ebp
pop     ebp
retn
main                endp

```

Практически, то же самое. Единственное что стоит отметить это то, что GCC решил зачем-то объединить вычисление `extended_model_id` и `extended_family_id` в один блок, вместо того чтобы вычислять их перед соответствующим вызовом `printf()`.

### Работа с типом `float` как со структурой

Как уже ранее указывалось в секции о FPU (?? (стр. ??)), и `float` и `double` содержат в себе *знак, мантиссу и экспоненту*. Однако, можем ли мы работать с этими полями напрямую? Попробуем с `float`.



( S — знак )

```

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <memory.h>

struct float_as_struct
{
    unsigned int fraction : 23; // мантисса
    unsigned int exponent : 8; // экспонента + 0x3FF
    unsigned int sign : 1; // бит знака
};

float f(float _in)
{
    float f=_in;
    struct float_as_struct t;

    assert (sizeof (struct float_as_struct) == sizeof (float));

    memcpy (&t, &f, sizeof (float));

    t.sign=1; // установить отрицательный знак
    t.exponent=t.exponent+2; // умножить d на 2^n (n здесь 2)
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

memcpy (&f, &t, sizeof (float));

return f;
};

int main()
{
printf ("%f\n", f(1.234));
};

```

Структура `float_as_struct` занимает в памяти столько же места сколько и `float`, то есть 4 байта или 32 бита.

Далее мы выставляем во входящем значении отрицательный знак, а также прибавляя двойку к экспоненте, мы тем самым умножаем всё значение на  $2^2$ , то есть на 4.

Компилируем в MSVC 2008 без включенной оптимизации:

#### Листинг 1.357: Неоптимизирующий MSVC 2008

```

_t$ = -8 ; size = 4
_f$ = -4 ; size = 4
_in$ = 8 ; size = 4
?f@@YAMM@Z PROC ; f
push ebp
mov ebp, esp
sub esp, 8

fld DWORD PTR __in$[ebp]
fstp DWORD PTR _f$[ebp]

push 4
lea eax, DWORD PTR _f$[ebp]
push eax
lea ecx, DWORD PTR _t$[ebp]
push ecx
call _memcpy
add esp, 12

mov edx, DWORD PTR _t$[ebp]
or edx, -2147483648 ; 80000000H - выставляем знак минус
mov DWORD PTR _t$[ebp], edx

mov eax, DWORD PTR _t$[ebp]
shr eax, 23 ; 00000017H - выкидываем мантиссу
and eax, 255 ; 000000ffH - оставляем здесь только экспоненту
add eax, 2 ; прибавляем к ней 2
and eax, 255 ; 000000ffH
shl eax, 23 ; 00000017H - поддвигаем результат на место бит
30:23
mov ecx, DWORD PTR _t$[ebp]
and ecx, -2139095041 ; 807fffffffH - выкидываем экспоненту

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; складываем оригинальное значение без экспоненты с новой только что
; вычисленной экспонентой:
or     ecx, eax
mov    DWORD PTR _t$[ebp], ecx

push  4
lea   edx, DWORD PTR _t$[ebp]
push  edx
lea   eax, DWORD PTR _f$[ebp]
push  eax
call  _memcpy
add   esp, 12

fld   DWORD PTR _f$[ebp]

mov   esp, ebp
pop   ebp
ret   0
?f@YAMM@Z ENDP ; f

```

Слегка избыточно. В версии скомпилированной с флагом /Ox нет вызовов memcpy(), там работа происходит сразу с переменной f. Но по неоптимизированной версии будет проще понять.

А что сделает GCC 4.4.1 с опцией -O3?

#### Листинг 1.358: Оптимизирующий GCC 4.4.1

```

; f(float)
public _Z1ff
_Z1ff proc near

var_4 = dword ptr -4
arg_0 = dword ptr 8

push  ebp
mov   ebp, esp
sub   esp, 4
mov   eax, [ebp+arg_0]
or    eax, 80000000h ; выставить знак минуса
mov   edx, eax
and   eax, 807FFFFFFh ; оставить в EAX только знак и мантиссу
shr   edx, 23 ; подготовить экспоненту
add   edx, 2 ; прибавить 2
movzx edx, dl ; сбросить все биты кроме 7:0 в EDX в 0
shl   edx, 23 ; подвинуть новую только что вычисленную
; экспоненту на свое место
or    eax, edx ; соединить новую экспоненту и оригинальное
; значение без экспоненты
mov   [ebp+var_4], eax
fld   [ebp+var_4]
leave
retn
_Z1ff endp

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

public main
main proc near
    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFFF0h
    sub     esp, 10h
    fld     ds:dword_8048614 ; -4.936
    fstp   qword ptr [esp+8]
    mov     dword ptr [esp+4], offset asc_8048610 ; "%f\n"
    mov     dword ptr [esp], 1
    call   ___printf_chk
    xor     eax, eax
    leave
    retn
main     endp

```

Да, функция `f()` в целом понятна. Однако, что интересно, еще при компиляции, невзирая на мешанину с полями структуры, GCC умудрился вычислить результат функции `f(1.234)` еще во время компиляции и сразу подставить его в аргумент для `printf()`!

### 1.30.7. Упражнения

- <http://challenges.re/71>
- <http://challenges.re/72>

## 1.31. Классическая ошибка с *struct*

Вот классическая ошибка с *struct*.

Простое определение:

```

struct test
{
    int field1;
    int field2;
};

```

И файлы на Си:

```

void setter(struct test *t, int a, int b)
{
    t->field1=a;
    t->field2=b;
};

```

```

#include <stdio.h>

void printer(struct test *t)
{

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    printf ("%d\n", t->field1);
    printf ("%d\n", t->field2);
};

```

Пока всё хорошо.

Теперь вы добавляете третье поле в структуру, где-то между двумя полями:

```

struct test
{
    int field1;
    int inserted;
    int field2;
};

```

И наверное вы модифицируете ф-цию `setter()`, но забываете про `printer()`:

```

void setter(struct test *t, int a, int b, int c)
{
    t->field1=a;
    t->inserted=b;
    t->field2=c;
};

```

Компилируете ваш проект, но файл на Си где находится ф-ция `printer()` не перекompилируется. потому что ваша **IDE!**<sup>165</sup> или система сборки не знает о том, что этот модуль зависим от определения структуры `test`. Может быть, потому что забыли добавить `#include <new.h>`. А может потому что заголовочный файл `new.h` включен в `printer.c` через какой-то другой заголовочный файл. Объектный файл остается неизменным (**IDE!** думает, что его не нужно пересобирать), в то время как ф-ция `setter()` уже новой версии. Эти два объектных файла (старый и новый) в итоге компонуются в исполняемый файл.

Потом вы его запускаете, и `setter()` записывает 3 поля по смещениям +0, +4 и +8. Хотя, ф-ция `printer()` знает только о двух полях, и во время печати читает их по смещениям +0 и +4.

Это приводит к очень запутанным и неприятным ошибкам. Причина в том, что **IDE!** или система сборки или `Makefile` не знают, что оба файла на Си (или модули) зависят от заголовочного файла с определением `test`. Популярное решение это пересобрать проект полностью, заново.

Это так же справедливо и для классов в Си++, так как они работают как структуры: ?? (стр. ??).

Это болезнь Си/Си++, и одна из причин критики, да. Множество новых **PL!** имеют лучшую поддержку модулей и интерфейсов. Но не забывайте о том, когда создавался компилятор Си: в 1970-ых, на старых компьютерах PDP. Так что создателями Си всё было упрощено до предела.

<sup>165</sup> **IDE!**

## 1.32. Объединения (`union`)

`union` в Си/Си++ используется в основном для интерпретации переменной (или блока памяти) одного типа как переменной другого типа.

### 1.32.1. Пример генератора случайных чисел

Если нам нужны случайные значения с плавающей запятой в интервале от 0 до 1, самое простое это взять **PRNG!**<sup>166</sup> вроде Mersenne twister. Он выдает случайные беззнаковые 32-битные числа (иными словами, он выдает 32 случайных бита). Затем мы можем преобразовать это число в `float` и затем разделить на `RAND_MAX` (`0xFFFFFFFF` в данном случае) — полученное число будет в интервале от 0 до 1.

Но как известно, операция деления — это медленная операция. Да и вообще хочется избежать лишних операций с FPU. Сможем ли мы избежать деления?

Вспомним состав числа с плавающей запятой: это бит знака, биты мантиссы и биты экспоненты. Для получения случайного числа, нам нужно просто заполнить случайными битами все биты мантиссы!

Экспонента не может быть нулевой (иначе число с плавающей точкой будет денормализованным), так что в эти биты мы запишем `0b01111111` — это будет означать что экспонента равна единице. Далее заполняем мантиссу случайными битами, знак оставляем в виде 0 (что значит наше число положительное), и вуаля. Генерируемые числа будут в интервале от 1 до 2, так что нам еще нужно будет отнять единицу.

В моем примере<sup>167</sup> применяется очень простой линейный конгруэнтный генератор случайных чисел, выдающий 32-битные числа. Генератор инициализируется текущим временем в стиле UNIX.

Далее, тип `float` представляется в виде `union` — это конструкция Си/Си++ позволяющая интерпретировать часть памяти по-разному. В нашем случае, мы можем создать переменную типа `union` и затем обращаться к ней как к `float` или как к `uint32_t`. Можно сказать, что это хак, причем грязный.

Код целочисленного **PRNG!** точно такой же, как мы уже рассматривали ранее: ?? (стр. ??). Так что и в скомпилированном виде этот код будет опущен.

```
#include <stdio.h>
#include <stdint.h>
#include <time.h>

// определения, данные и ф-ции для целочисленного PRNG

// константы из книги Numerical Recipes
const uint32_t RNG_a=1664525;
const uint32_t RNG_c=1013904223;
uint32_t RNG_state; // глобальная переменная
```

<sup>166</sup> **PRNG!**

<sup>167</sup> идея взята отсюда: [URL](#)



```

void my_srand(uint32_t i)
{
    RNG_state=i;
};

uint32_t my_rand()
{
    RNG_state=RNG_state*RNG_a+RNG_c;
    return RNG_state;
};

// определения и ф-ции FPU PRNG:

union uint32_t_float
{
    uint32_t i;
    float f;
};

float float_rand()
{
    union uint32_t_float tmp;
    tmp.i=my_rand() & 0x007ffffff | 0x3F800000;
    return tmp.f-1;
};

// тест

int main()
{
    my_srand(time(NULL)); // инициализация PRNG

    for (int i=0; i<100; i++)
        printf ("%f\n", float_rand());

    return 0;
};

```

**x86**

Листинг 1.359: Оптимизирующий MSVC 2010

```

$SG4238 DB      '%f', 0aH, 00H

__real@3ff0000000000000 DQ 03ff000000000000r    ; 1

tv130 = -4
_tmp$ = -4
?float_rand@@YAMXZ PROC
    push    ecx
    call    ?my_rand@@YAIXZ
; EAX=псевдослучайное значение

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    and    eax, 8388607          ; 007ffffFH
    or     eax, 1065353216      ; 3f800000H
; EAX=псевдослучайное значение & 0x007ffff | 0x3f800000
; сохранить его в локальном стеке
    mov    DWORD PTR _tmp$[esp+4], eax
; перезагрузить его как число с плавающей точкой:
    fld   DWORD PTR _tmp$[esp+4]
; вычесть 1.0:
    fsub  QWORD PTR __real@3ff0000000000000
; сохранить полученное значение в локальном стеке и перезагрузить его
; эти инструкции избыточны:
    fstp  DWORD PTR tv130[esp+4]
    fld   DWORD PTR tv130[esp+4]
    pop   ecx
    ret   0
?float_rand@@YAMXZ ENDP

_main  PROC
    push  esi
    xor   eax, eax
    call  _time
    push  eax
    call  ?my_srand@@YAXI@Z
    add   esp, 4
    mov   esi, 100
$LL3@main:
    call  ?float_rand@@YAMXZ
    sub   esp, 8
    fstp  QWORD PTR [esp]
    push  OFFSET $SG4238
    call  _printf
    add   esp, 12
    dec   esi
    jne   SHORT $LL3@main
    xor   eax, eax
    pop   esi
    ret   0
_main  ENDP

```

Имена функций такие странные, потому что этот пример был скомпилирован как Си++, и это манглинг имен в Си++, мы будем рассматривать это позже: ?? (стр. ??).

Если скомпилировать это в MSVC 2012, компилятор будет использовать SIMD-инструкции для FPU, читайте об этом здесь:

?? (стр. ??).

## ARM (Режим ARM)

Листинг 1.360: Оптимизирующий GCC 4.6.3 (IDA)

```
float_rand
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

                STMFD  SP!, {R3,LR}
                BL     my_rand
; R0=псевдослучайное значение
                FLDS   S0, =1.0
; S0=1.0
                BIC   R3, R0, #0xFF000000
                BIC   R3, R3, #0x800000
                ORR   R3, R3, #0x3F800000
; R3=псевдослучайное значение & 0x007fffff | 0x3f800000
; копировать из R3 в FPU (регистр S15).
; это работает как побитовое копирование, без всякого конвертирования
                FMSR   S15, R3
; вычесть 1.0 и оставить результат в S0:
                FSUBS  S0, S15, S0
                LDMFD  SP!, {R3,PC}

flt_5C          DCFS 1.0

main
                STMFD  SP!, {R4,LR}
                MOV    R0, #0
                BL     time
                BL     my_srand
                MOV    R4, #0x64 ; 'd'

loc_78
                BL     float_rand
; S0=псевдослучайное значение
                LDR    R0, =aF          ; "%f"
; сконвертировать значение типа float в значение типа double (это нужно для
printf()):
                FCVTDS D7, S0
; побитовое копирование из D7 в пару регистров R2/R3 (для printf()):
                FMRRD  R2, R3, D7
                BL     printf
                SUBS   R4, R4, #1
                BNE   loc_78
                MOV    R0, R4
                LDMFD  SP!, {R4,PC}

aF              DCB "%f",0xA,0

```

Мы также сделаем дамп в objdump и увидим, что FPU-инструкции имеют немного другие имена чем в **IDA!** Наверное, разработчики IDA и binutils пользовались разной документацией? Должно быть, будет полезно знать оба варианта названий инструкций.

Листинг 1.361: Оптимизирующий GCC 4.6.3 (objdump)

```

00000038 <float_rand>:
38:  e92d4008      push    {r3, lr}
3c:  ebfffffe      bl     10 <my_rand>
40:  ed9f0a05      vldr   s0, [pc, #20] ; 5c <float_rand+0x24>
44:  e3c034ff      bic    r3, r0, #-16777216 ; 0xff000000

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

48: e3c33502    bic    r3, r3, #8388608    ; 0x800000
4c: e38335fe    orr    r3, r3, #1065353216 ; 0x3f800000
50: ee073a90    vmov   s15, r3
54: ee370ac0    vsub.f32    s0, s15, s0
58: e8bd8008    pop    {r3, pc}
5c: 3f800000    svccc  0x00800000

00000000 <main>:
 0: e92d4010    push   {r4, lr}
 4: e3a00000    mov    r0, #0
 8: ebfffffe    bl     0 <time>
 c: ebfffffe    bl     0 <main>
10: e3a04064    mov    r4, #100            ; 0x64
14: ebfffffe    bl     38 <main+0x38>
18: e59f0018    ldr    r0, [pc, #24]      ; 38 <main+0x38>
1c: eeb77ac0    vcvf.f64.f32    d7, s0
20: ec532b17    vmov   r2, r3, d7
24: ebfffffe    bl     0 <printf>
28: e2544001    subs   r4, r4, #1
2c: lafffff8    bne   14 <main+0x14>
30: e1a00004    mov    r0, r4
34: e8bd8010    pop    {r4, pc}
38: 00000000    andeq  r0, r0, r0

```

Инструкции по адресам 0x5c в `float_rand()` и 0x38 в `main()` это (псевдо-)случайный мусор.

### 1.32.2. Вычисление машинного эпсилона

Машинный эпсилон — это самая маленькая гранула, с которой может работать **FPU!**<sup>168</sup>. Чем больше бит выделено для числа с плавающей точкой, тем меньше машинный эпсилон. Это  $2^{-23} \approx 1.19e-07$  для *float* и  $2^{-52} \approx 2.22e-16$  для *double*. См. также: [статью в Wikipedia](#).

Любопытно, что вычислить машинный эпсилон очень легко:

```

#include <stdio.h>
#include <stdint.h>

union uint_float
{
    uint32_t i;
    float f;
};

float calculate_machine_epsilon(float start)
{
    union uint_float v;
    v.f=start;
    v.i++;
    return v.f-start;
}

```

<sup>168</sup>В русскоязычной литературе встречается также термин «машинный ноль».

```

}

void main()
{
    printf ("%g\n", calculate_machine_epsilon(1.0));
};

```

Что мы здесь делаем это обходимся с мантиссой числа в формате IEEE 754 как с целочисленным числом и прибавляем единицу к нему. Итоговое число с плавающей точкой будет равно  $starting\_value + machine\_epsilon$ , так что нам нужно просто вычесть изначальное значение (используя арифметику с плавающей точкой) чтобы измерить, какое число отражает один бит в одинарной точности (*float*). *union* здесь нужен чтобы мы могли обращаться к числу в формате IEEE 754 как к обычному целочисленному. Прибавление 1 к нему на самом деле прибавляет 1 к *мантиссе* числа, хотя, нужно сказать, переполнение также возможно, что приведет к прибавлению единицы к экспоненте.

## x86

Листинг 1.362: Оптимизирующий MSVC 2010

```

tv130 = 8
_v$ = 8
_start$ = 8
_calculate_machine_epsilon PROC
    fld     DWORD PTR _start$[esp-4]
; это лишняя инструкция:
    fst     DWORD PTR _v$[esp-4]
    inc     DWORD PTR _v$[esp-4]
    fsubr   DWORD PTR _v$[esp-4]
; эта пара инструкций также лишняя:
    fstp    DWORD PTR tv130[esp-4]
    fld     DWORD PTR tv130[esp-4]
    ret     0
_calculate_machine_epsilon ENDP

```

Вторая инструкция FST избыточная: нет необходимости сохранять входное значение в этом же месте (компилятор решил выделить переменную *v* в том же месте локального стека, где находится и входной аргумент). Далее оно инкрементируется при помощи INC, как если это обычная целочисленная переменная. Затем оно загружается в FPU как если это 32-битное число в формате IEEE 754, FSUBR делает остальную часть работы и результат в ST0. Последняя пара инструкций FSTP/FLD избыточна, но компилятор не оптимизировал её.

## ARM64

Расширим этот пример до 64-бит:

```

#include <stdio.h>
#include <stdint.h>

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

typedef union
{
    uint64_t i;
    double d;
} uint_double;

double calculate_machine_epsilon(double start)
{
    uint_double v;
    v.d=start;
    v.i++;
    return v.d-start;
}

void main()
{
    printf ("%g\n", calculate_machine_epsilon(1.0));
};

```

В ARM64 нет инструкции для добавления числа к D-регистру в FPU, так что входное значение (пришедшее в D0) в начале копируется в **GPR!**, инкрементируется, копируется в регистр FPU D1, затем происходит вычитание.

Листинг 1.363: Оптимизирующий GCC 4.9 ARM64

```

calculate_machine_epsilon:
    fmov    x0, d0      ; загрузить входное значение типа double в X0
    add    x0, x0, 1    ; X0++
    fmov    d1, x0      ; переместить его в регистр FPU
    fsub   d0, d1, d0   ; вычесть
    ret

```

Смотрите также этот пример скомпилированный под x64 с SIMD-инструкциями: ?? (стр. ??).

## MIPS

Новая для нас здесь инструкция это MTC1 («Move To Coprocessor 1»), она просто переносит данные из **GPR!** в регистры FPU.

Листинг 1.364: Оптимизирующий GCC 4.4.5 (IDA)

```

calculate_machine_epsilon:
    mfc1    $v0, $f12
    or     $at, $zero ; NOP
    addiu  $v1, $v0, 1
    mtc1   $v1, $f2
    jr     $ra
    sub.s  $f0, $f2, $f12 ; branch delay slot

```

## Вывод

Трудно сказать, понадобится ли кому-то такая эквилибристика в реальном коде, но как уже было упомянуто много раз в этой книге, этот пример хорошо подходит для объяснения формата IEEE 754 и *union* в Си/Си++.

### 1.32.3. Замена инструкции FSCALE

Agner Fog в своей работе *Optimizing subroutines in assembly language / An optimization guide for x86 platforms*<sup>169</sup> указывает, что инструкция **FPU!** FSCALE (вычисление  $2^n$ ) может быть медленной на многих CPU, и он предлагает более быструю замену.

Вот мой перевод его кода на ассемблер на Си/Си++:

```
#include <stdint.h>
#include <stdio.h>

union uint_float
{
    uint32_t i;
    float f;
};

float flt_2n(int N)
{
    union uint_float tmp;

    tmp.i=(N<<23)+0x3f800000;
    return tmp.f;
};

struct float_as_struct
{
    unsigned int fraction : 23;
    unsigned int exponent : 8;
    unsigned int sign : 1;
};

float flt_2n_v2(int N)
{
    struct float_as_struct tmp;

    tmp.fraction=0;
    tmp.sign=0;
    tmp.exponent=N+0x7f;
    return *(float*)&tmp;
};

union uint64_double
{
    uint64_t i;
```

<sup>169</sup>[http://www.agner.org/optimize/optimizing\\_assembly.pdf](http://www.agner.org/optimize/optimizing_assembly.pdf)

```

    double d;
};

double dbl_2n(int N)
{
    union uint64_double tmp;

    tmp.i=((uint64_t)N<<52)+0x3ff0000000000000UL;
    return tmp.d;
};

struct double_as_struct
{
    uint64_t fraction : 52;
    int exponent : 11;
    int sign : 1;
};

double dbl_2n_v2(int N)
{
    struct double_as_struct tmp;

    tmp.fraction=0;
    tmp.sign=0;
    tmp.exponent=N+0x3ff;
    return *(double*)&tmp;
};

int main()
{
    // 211 = 2048
    printf ("%f\n", flt_2n(11));
    printf ("%f\n", flt_2n_v2(11));
    printf ("%lf\n", dbl_2n(11));
    printf ("%lf\n", dbl_2n_v2(11));
};

```

Инструкция FSCALE в вашей среде может быть быстрее, но всё же, это хорошая демонстрация *union*-а и того факта, что экспонента хранится в виде  $2^n$ , так что входное значение  $n$  сдвигается в экспоненту закодированного в IEEE 754 числа. Потом экспонента корректируется прибавлением  $0x3f800000$  или  $0x3ff0000000000000$ .

То же самое можно сделать без сдвигов, при помощи *struct*, но всё равно, внутри будет операция.

### 1.32.4. Быстрое вычисление квадратного корня

Вот где еще можно на практике применить трактовку типа *float* как целочисленного, это быстрое вычисление квадратного корня.



Листинг 1.365: Исходный код взят из Wikipedia: [https://en.wikipedia.org/wiki/Methods\\_of\\_computing\\_square\\_roots](https://en.wikipedia.org/wiki/Methods_of_computing_square_roots)

```

/* Assumes that float is in the IEEE 754 single precision floating point ↵
   ↵ format
   * and that int is 32 bits. */
float sqrt_approx(float z)
{
    int val_int = *(int*)&z; /* Same bits, but as an int */
    /*
     * To justify the following code, prove that
     *
     * (((val_int / 2^m) - b) / 2) + b) * 2^m = ((val_int - 2^m) / 2) + ((↵
     ↵ b + 1) / 2) * 2^m)
     *
     * where
     *
     * b = exponent bias
     * m = number of mantissa bits
     *
     * .
     */

    val_int -= 1 << 23; /* Subtract 2^m. */
    val_int >>= 1; /* Divide by 2. */
    val_int += 1 << 29; /* Add ((b + 1) / 2) * 2^m. */

    return *(float*)&val_int; /* Interpret again as float */
}

```

В качестве упражнения, вы можете попробовать скомпилировать эту функцию и разобраться, как она работает.

Имеется также известный алгоритм быстрого вычисления  $\frac{1}{\sqrt{x}}$ . Алгоритм стал известным, вероятно потому, что был применен в Quake III Arena.

Описание алгоритма есть в Wikipedia: [https://ru.wikipedia.org/wiki/Быстрый\\_обратный\\_квадратный\\_корень](https://ru.wikipedia.org/wiki/Быстрый_обратный_квадратный_корень).

## 1.33. Указатели на функции

Указатель на функцию, в целом, как и любой другой указатель, просто адрес, указывающий на начало функции в сегменте кода.

Это часто применяется для вызовов т.н. callback-функций.

Известные примеры:

- `qsort()`, `atexit()` из стандартной библиотеки Си;
- сигналы в \*NIX ОС;
- запуск тредов: `CreateThread()` (win32), `pthread_create()` (POSIX);

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

- множество функций win32, например EnumChildWindows().
- множество мест в ядре Linux, например, функции драйверов файловой системы вызываются через callback-и.
- функции плагинов GCC также вызываются через callback-и.

Итак, функция `qsort()` это реализация алгоритма «быстрой сортировки». Функция может сортировать что угодно, любые типы данных, но при условии, что вы имеете функцию сравнения этих двух элементов данных и `qsort()` может вызывать её.

Эта функция сравнения может определяться так:

```
int (*compare)(const void *, const void *)
```

Попробуем такой пример:

```
1 /* ex3 Sorting ints with qsort */
2
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int comp(const void * _a, const void * _b)
7 {
8     const int *a=(const int *)_a;
9     const int *b=(const int *)_b;
10
11     if (*a==*b)
12         return 0;
13     else
14         if (*a < *b)
15             return -1;
16         else
17             return 1;
18 }
19
20 int main(int argc, char* argv[])
21 {
22     int numbers[10]={1892,45,200,-98,4087,5,-12345,1087,88,-100000};
23     int i;
24
25     /* Sort the array */
26     qsort(numbers,10,sizeof(int),comp) ;
27     for (i=0;i<9;i++)
28         printf("Number = %d\n",numbers[ i ] ) ;
29     return 0;
30 }
```

### 1.33.1. MSVC

Компилируем в MSVC 2010 (некоторые части убраны для краткости) с опцией `/Ox`:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Листинг 1.366: Оптимизирующий MSVC 2010: /GS- /MD

```

__a$ = 8 ; size = 4
__b$ = 12 ; size = 4
_comp PROC
    mov     eax, DWORD PTR __a$[esp-4]
    mov     ecx, DWORD PTR __b$[esp-4]
    mov     eax, DWORD PTR [eax]
    mov     ecx, DWORD PTR [ecx]
    cmp     eax, ecx
    jne     SHORT $LN4@comp
    xor     eax, eax
    ret     0
$LN4@comp:
    xor     edx, edx
    cmp     eax, ecx
    setge   dl
    lea    eax, DWORD PTR [edx+edx-1]
    ret     0
_comp ENDP

_numbers$ = -40 ; size = 40
_argc$ = 8 ; size = 4
_argv$ = 12 ; size = 4
_main PROC
    sub     esp, 40 ; 00000028H
    push   esi
    push   OFFSET _comp
    push   4
    lea    eax, DWORD PTR _numbers$[esp+52]
    push   10 ; 0000000aH
    push   eax
    mov    DWORD PTR _numbers$[esp+60], 1892 ; 00000764H
    mov    DWORD PTR _numbers$[esp+64], 45 ; 0000002dH
    mov    DWORD PTR _numbers$[esp+68], 200 ; 000000c8H
    mov    DWORD PTR _numbers$[esp+72], -98 ; ffffffff9eH
    mov    DWORD PTR _numbers$[esp+76], 4087 ; 00000ff7H
    mov    DWORD PTR _numbers$[esp+80], 5
    mov    DWORD PTR _numbers$[esp+84], -12345 ; ffffcfc7H
    mov    DWORD PTR _numbers$[esp+88], 1087 ; 0000043fH
    mov    DWORD PTR _numbers$[esp+92], 88 ; 00000058H
    mov    DWORD PTR _numbers$[esp+96], -100000 ; fffe7960H
    call   _qsort
    add    esp, 16 ; 00000010H
    ...

```

Ничего особо удивительного здесь мы не видим. В качестве четвертого аргумента, в `qsort()` просто передается адрес метки `_comp`, где собственно и располагается функция `comp()`, или, можно сказать, самая первая инструкция этой функции.

Как `qsort()` вызывает её?

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Посмотрим в MSVCR80.DLL (эта DLL куда в MSVC вынесены функции из стандартных библиотек Си):

Листинг 1.367: MSVCR80.DLL

```
.text:7816CBF0 ; void __cdecl qsort(void *, unsigned int, unsigned int, int
    (__cdecl *)(const void *, const void *))
.text:7816CBF0         public _qsort
.text:7816CBF0         _qsort          proc near
.text:7816CBF0         lo              = dword ptr -104h
.text:7816CBF0         hi              = dword ptr -100h
.text:7816CBF0         var_FC         = dword ptr -0FCh
.text:7816CBF0         stkptr         = dword ptr -0F8h
.text:7816CBF0         lostk         = dword ptr -0F4h
.text:7816CBF0         histk         = dword ptr -7Ch
.text:7816CBF0         base          = dword ptr 4
.text:7816CBF0         num           = dword ptr 8
.text:7816CBF0         width        = dword ptr 0Ch
.text:7816CBF0         comp          = dword ptr 10h
.text:7816CBF0
.text:7816CBF0         sub          esp, 100h

....

.text:7816CCE0 loc_7816CCE0:                                ; CODE XREF: _qsort+B1
.text:7816CCE0         shr          eax, 1
.text:7816CCE2         imul         eax, ebp
.text:7816CCE5         add          eax, ebx
.text:7816CCE7         mov          edi, eax
.text:7816CCE9         push        edi
.text:7816CCEA         push        ebx
.text:7816CCEB         call        [esp+118h+comp]
.text:7816CCF2         add          esp, 8
.text:7816CCF5         test        eax, eax
.text:7816CCF7         jle         short loc_7816CD04
```

comp — это четвертый аргумент функции. Здесь просто передается управление по адресу, указанному в comp. Перед этим подготавливается два аргумента для функции comp(). Далее, проверяется результат её выполнения.

Вот почему использование указателей на функции — это опасно. Во-первых, если вызвать qsort() с неправильным указателем на функцию, то qsort(), дойдя до этого вызова, может передать управление неизвестно куда, процесс упадет, и эту ошибку можно будет найти не сразу.

Во-вторых, типизация callback-функции должна строго соблюдаться, вызов не той функции с не теми аргументами не того типа, может привести к плачевным результатам, хотя падение процесса это и не проблема, проблема — это найти ошибку, ведь компилятор на стадии компиляции может вас и не предупредить о потенциальных неприятностях.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

## MSVC + OllyDbg

Загрузим наш пример в OllyDbg и установим точку останова на функции `comp()`. Как значения сравниваются, мы можем увидеть во время самого первого вызова `comp()`:

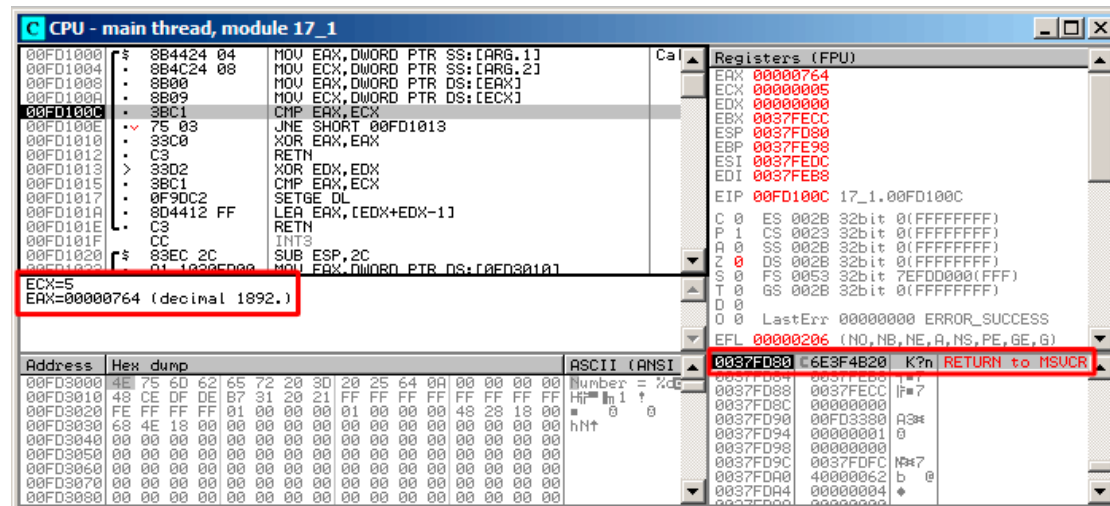


Рис. 1.109: OllyDbg: первый вызов `comp()`

Для удобства, OllyDbg показывает сравниваемые значения в окне под окном кода. Мы можем так же увидеть, что **SP!** указывает на **RA!** где находится место в функции `qsort()` (на самом деле, находится в `MSVCRT100.DLL`).

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Трассируя (F8) до инструкции RETN и нажав F8 еще один раз, мы возвращаемся в функцию qsort():

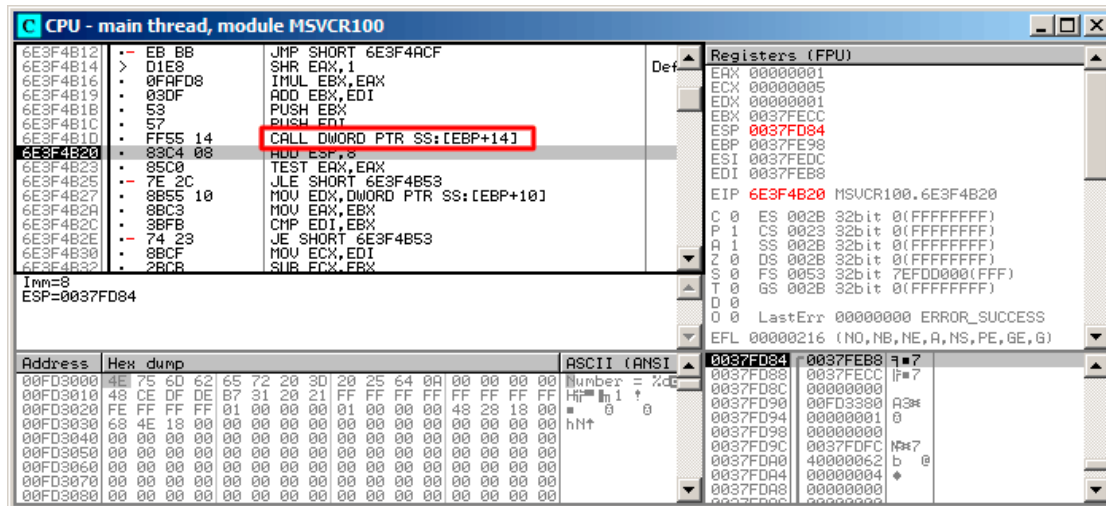


Рис. 1.110: OllyDbg: код в qsort() сразу после вызова comp()

Это был вызов функции сравнения.

Вот также скриншот момента второго вызова функции `comp()` — теперь сравниваемые значения другие:

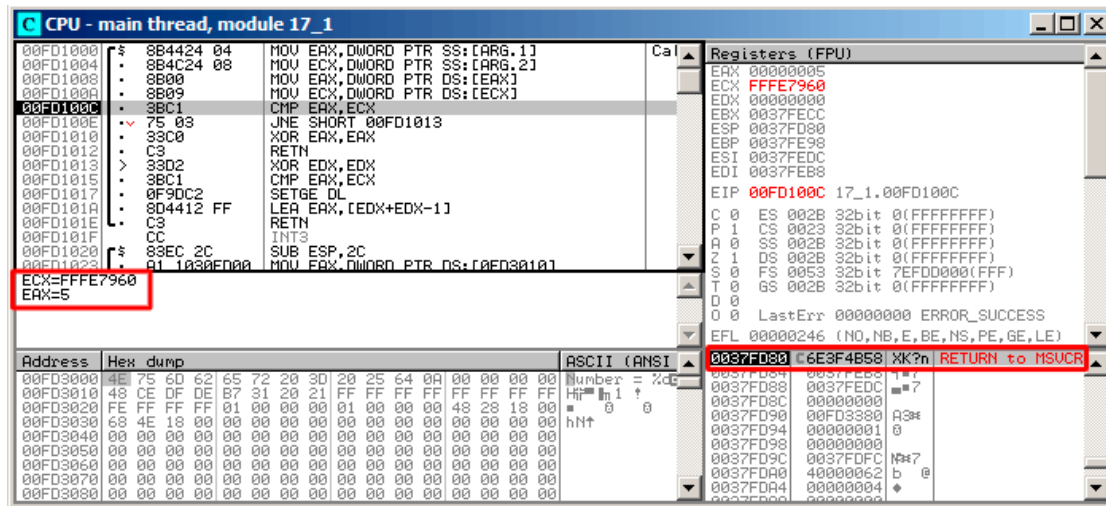


Рис. 1.111: OllyDbg: второй вызов `comp()`

## MSVC + tracer

Посмотрим, какие пары сравниваются. Эти 10 чисел будут сортироваться: 1892, 45, 200, -98, 4087, 5, -12345, 1087, 88, -100000.

Найдем адрес первой инструкции `CMP` в `comp()` и это `0x0040100C` и мы ставим точку останова на ней:

```
tracer.exe -l:17_1.exe bpx=17_1.exe!0x0040100C
```

Получаем информацию о регистрах на точке останова:

```
PID=4336|New process 17_1.exe
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=IF
(0) 17_1.exe!0x40100c
EAX=0x00000005 EBX=0x0051f7c8 ECX=0xfffe7960 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=PF ZF IF
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=CF PF ZF IF
...
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Отфильтруем EAX и ECX и получим:

```
EAX=0x00000764 ECX=0x00000005
EAX=0x00000005 ECX=0xfffe7960
EAX=0x00000764 ECX=0x00000005
EAX=0x0000002d ECX=0x00000005
EAX=0x00000058 ECX=0x00000005
EAX=0x0000043f ECX=0x00000005
EAX=0xffffcfc7 ECX=0x00000005
EAX=0x000000c8 ECX=0x00000005
EAX=0xffffffff9e ECX=0x00000005
EAX=0x00000ff7 ECX=0x00000005
EAX=0x00000ff7 ECX=0x00000005
EAX=0xffffffff9e ECX=0x00000005
EAX=0xffffffff9e ECX=0x00000005
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0x00000005 ECX=0xffffcfc7
EAX=0xffffffff9e ECX=0x00000005
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0xffffffff9e ECX=0xffffcfc7
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0x000000c8 ECX=0x00000ff7
EAX=0x0000002d ECX=0x00000ff7
EAX=0x0000043f ECX=0x00000ff7
EAX=0x00000058 ECX=0x00000ff7
EAX=0x00000764 ECX=0x00000ff7
EAX=0x000000c8 ECX=0x00000764
EAX=0x0000002d ECX=0x00000764
EAX=0x0000043f ECX=0x00000764
EAX=0x00000058 ECX=0x00000764
EAX=0x000000c8 ECX=0x00000058
EAX=0x0000002d ECX=0x000000c8
EAX=0x0000043f ECX=0x000000c8
EAX=0x000000c8 ECX=0x00000058
EAX=0x0000002d ECX=0x000000c8
EAX=0x0000002d ECX=0x00000058
```

Это 34 пары. Следовательно, алгоритму быстрой сортировки нужно 34 операции сравнения для сортировки этих 10-и чисел.



## MSVC + tracer (code coverage)

Но можно также и воспользоваться возможностью tracer накапливать все возможные состояния регистров и показать их в **IDA!**.

Трассируем все инструкции в функции `comp()`:

```
tracer.exe -l:17_1.exe bpf=17_1.exe!0x00401000,trace:cc
```

Получем .idc-скрипт для загрузки в **IDA!** и загружаем его:

```
.text:00401000
.text:00401000 ; int __cdecl PtFuncCompare(const void *, const void *)
.text:00401000 PtFuncCompare proc near ; DATA XREF: _main+5↓o
.text:00401000
.text:00401000 arg_0 = dword ptr 4
.text:00401000 arg_4 = dword ptr 8
.text:00401000
.text:00401000 mov eax, [esp+arg_0] ; [ESP+4]=0x45f7ec..0x45f810(step=4), L"?\x04?
.text:00401004 mov ecx, [esp+arg_4] ; [ESP+8]=0x45f7ec..0x45f7f4(step=4), 0x45f7fc
.text:00401008 mov eax, [eax] ; [EAX]=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff
.text:0040100a mov ecx, [ecx] ; [ECX]=5, 0x58, 0xc8, 0x764, 0xff7, 0xfffe7960
.text:0040100c cmp eax, ecx ; EAX=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff7,
.text:0040100e jnz short loc_401013 ; ZF=false
.text:00401010 xor eax, eax
.text:00401012 retn
.text:00401013 ; -----
.text:00401013
.text:00401013 loc_401013: ; CODE XREF: PtFuncCompare+E↑j
.text:00401013 xor edx, edx
.text:00401015 cmp eax, ecx ; EAX=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff7,
.text:00401017 setnl dl ; SF=false,true OF=false
.text:0040101a lea eax, [edx+edx-1]
.text:0040101e retn ; EAX=1, 0xffffffff
.text:0040101e PtFuncCompare endp
.text:0040101f
```

Рис. 1.112: tracer и IDA. N.B.: некоторые значения обрезаны справа

Имя этой функции (`PtFuncCompare`) дала **IDA!**— видимо, потому что видит что указатель на эту функцию передается в `qsort()`.

Мы видим, что указатели *a* и *b* указывают на разные места внутри массива, но шаг между указателями — 4, что логично, ведь в массиве хранятся 32-битные значения.

Видно, что инструкции по адресам `0x401010` и `0x401012` никогда не исполнялись (они и остались белыми): действительно, функция `comp()` никогда не возвращала 0, потому что в массиве нет одинаковых элементов.

### 1.33.2. GCC

Не слишком большая разница:

Листинг 1.368: GCC

```
lea    eax, [esp+40h+var_28]
mov    [esp+40h+var_40], eax
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov     [esp+40h+var_28], 764h
mov     [esp+40h+var_24], 2Dh
mov     [esp+40h+var_20], 0C8h
mov     [esp+40h+var_1C], 0FFFFFF9Eh
mov     [esp+40h+var_18], 0FF7h
mov     [esp+40h+var_14], 5
mov     [esp+40h+var_10], 0FFFCFC7h
mov     [esp+40h+var_C], 43Fh
mov     [esp+40h+var_8], 58h
mov     [esp+40h+var_4], 0FFFE7960h
mov     [esp+40h+var_34], offset comp
mov     [esp+40h+var_38], 4
mov     [esp+40h+var_3C], 0Ah
call    _qsort

```

Функция comp():

```

public comp
comp      proc near
arg_0     = dword ptr 8
arg_4     = dword ptr 0Ch

        push    ebp
        mov     ebp, esp
        mov     eax, [ebp+arg_4]
        mov     ecx, [ebp+arg_0]
        mov     edx, [eax]
        xor     eax, eax
        cmp     [ecx], edx
        jnz     short loc_8048458
        pop     ebp
        retn

loc_8048458:
        setnl   al
        movzx   eax, al
        lea    eax, [eax+eax-1]
        pop     ebp
        retn

comp      endp

```

Реализация qsort() находится в libc.so.6, и представляет собой просто wrapper <sup>170</sup> для qsort\_r().

Она, в свою очередь, вызывает quicksort(), где есть вызовы определенной нами функции через переданный указатель:

Листинг 1.369: (файл libc.so.6, версия glibc: 2.10.1)

```

.text:0002DDF6      mov     edx, [ebp+arg_10]
.text:0002DDF9      mov     [esp+4], esi
.text:0002DDFD      mov     [esp], edi

```

<sup>170</sup>понятие близкое к [thunk function](#)

```
.text:0002DE00      mov     [esp+8], edx
.text:0002DE04      call   [ebp+arg_C]
...
```

### **GCC + GDB (с исходными кодами)**

Очевидно, у нас есть исходный код нашего примера на Си (?? (стр. ??)), так что мы можем установить точку останова (*b*) на номере строки (11-й — это номер строки где происходит первое сравнение). Нам также нужно скомпилировать наш пример с ключом `-g`, чтобы в исполняемом файле была полная отладочная информация. Мы можем так же выводить значения используя имена переменных (*p*): отладочная информация также содержит информацию о том, в каком регистре и/или элементе локального стека находится какая переменная.

Мы можем также увидеть стек (*bt*) и обнаружить что в Glibc используется какая-то вспомогательная функция с именем `msort_with_tmp()`.

Листинг 1.370: GDB-сессия

```
dennis@ubuntuv:~/polygon$ gcc 17_1.c -g
dennis@ubuntuv:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
...
Reading symbols from /home/dennis/polygon/a.out...done.
(gdb) b 17_1.c:11
Breakpoint 1 at 0x804845f: file 17_1.c, line 11.
(gdb) run
Starting program: /home/dennis/polygon/./a.out

Breakpoint 1, comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c:11
11      if (*a==*b)
(gdb) p *a
$1 = 1892
(gdb) p *b
$2 = 45
(gdb) c
Continuing.

Breakpoint 1, comp (_a=0xbffff104, _b=_b@entry=0xbffff108) at 17_1.c:11
11      if (*a==*b)
(gdb) p *a
$3 = -98
(gdb) p *b
$4 = 4087
(gdb) bt
#0  comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c:11
#1  0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0
    ↪ xbffff0f8, n=n@entry=2)
    at msort.c:65
#2  0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0xbffff07c) at msort.c:
    ↪ 45
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

#3 msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry↵
↳ =5) at msort.c:53
#4 0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0xbffff07c) at msort↵
↳ .c:45
#5 msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry↵
↳ =10) at msort.c:53
#6 0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0xbffff07c) at ↵
↳ msort.c:45
#7 __GI_qsorth_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry=4, cmp=↵
↳ cmp@entry=0x804844d <comp>,
arg=arg@entry=0x0) at msort.c:297
#8 0xb7e42dcf in __GI_qsorth (b=0xbffff0f8, n=10, s=4, cmp=0x804844d <comp↵
↳ >) at msort.c:307
#9 0x0804850d in main (argc=1, argv=0xbffff1c4) at 17_1.c:26
(gdb)

```

### GCC + GDB (без исходных кодов)

Но часто никаких исходных кодов нет вообще, так что мы можем дизассемблировать функцию `comp()` (`disas`), найти самую первую инструкцию `CMPL` и установить точку останова (`b`) по этому адресу. На каждой точке останова мы будем видеть содержимое регистров (`info registers`). Информация из стека так же доступна (`bt`), но частичная: здесь нет номеров строк для функции `comp()`.

#### Листинг 1.371: GDB-сессия

```

dennis@ubuntuvm:~/polygon$ gcc 17_1.c
dennis@ubuntuvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
...
Reading symbols from /home/dennis/polygon/a.out...(no debugging symbols ↵
↳ found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas comp
Dump of assembler code for function comp:
   0x0804844d <+0>:    push    ebp
   0x0804844e <+1>:    mov     ebp,esp
   0x08048450 <+3>:    sub     esp,0x10
   0x08048453 <+6>:    mov     eax,DWORD PTR [ebp+0x8]
   0x08048456 <+9>:    mov     DWORD PTR [ebp-0x8],eax
   0x08048459 <+12>:   mov     eax,DWORD PTR [ebp+0xc]
   0x0804845c <+15>:   mov     DWORD PTR [ebp-0x4],eax
   0x0804845f <+18>:   mov     eax,DWORD PTR [ebp-0x8]
   0x08048462 <+21>:   mov     edx,DWORD PTR [eax]
   0x08048464 <+23>:   mov     eax,DWORD PTR [ebp-0x4]
   0x08048467 <+26>:   mov     eax,DWORD PTR [eax]
   0x08048469 <+28>:   cmp     edx,eax
   0x0804846b <+30>:   jne    0x8048474 <comp+39>
   0x0804846d <+32>:   mov     eax,0x0
   0x08048472 <+37>:   jmp    0x804848e <comp+65>
   0x08048474 <+39>:   mov     eax,DWORD PTR [ebp-0x8]
   0x08048477 <+42>:   mov     edx,DWORD PTR [eax]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

0x08048479 <+44>:  mov    eax,DWORD PTR [ebp-0x4]
0x0804847c <+47>:  mov    eax,DWORD PTR [eax]
0x0804847e <+49>:  cmp    edx,eax
0x08048480 <+51>:  jge   0x8048489 <comp+60>
0x08048482 <+53>:  mov    eax,0xffffffff
0x08048487 <+58>:  jmp   0x804848e <comp+65>
0x08048489 <+60>:  mov    eax,0x1
0x0804848e <+65>:  leave
0x0804848f <+66>:  ret

```

End of assembler dump.

(gdb) b \*0x08048469

Breakpoint 1 at 0x8048469

(gdb) run

Starting program: /home/dennis/polygon/./a.out

Breakpoint 1, 0x08048469 in comp ()

(gdb) info registers

```

eax          0x2d      45
ecx          0xbffff0f8  -1073745672
edx          0x764     1892
ebx          0xb7fc0000  -1208221696
esp          0xbffffeeb8  0xbffffeeb8
ebp          0xbffffeec8  0xbffffeec8
esi          0xbffff0fc  -1073745668
edi          0xbffff010  -1073745904
eip          0x8048469  0x8048469 <comp+28>
eflags      0x286     [ PF SF IF ]
cs          0x73     115
ss          0x7b     123
ds          0x7b     123
es          0x7b     123
fs          0x0      0
gs          0x33     51

```

(gdb) c

Continuing.

Breakpoint 1, 0x08048469 in comp ()

(gdb) info registers

```

eax          0xff7     4087
ecx          0xbffff104  -1073745660
edx          0xffffffff9e  -98
ebx          0xb7fc0000  -1208221696
esp          0xbffffee58  0xbffffee58
ebp          0xbffffee68  0xbffffee68
esi          0xbffff108  -1073745656
edi          0xbffff010  -1073745904
eip          0x8048469  0x8048469 <comp+28>
eflags      0x282     [ SF IF ]
cs          0x73     115
ss          0x7b     123
ds          0x7b     123
es          0x7b     123
fs          0x0      0

```

```

gs          0x33      51
(gdb) c
Continuing.

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax          0xffffffff9e      -98
ecx          0xbffff100    -1073745664
edx          0xc8        200
ebx          0xb7fc0000    -1208221696
esp          0xbffffeeb8    0xbffffeeb8
ebp          0xbffffeec8    0xbffffeec8
esi          0xbffff104    -1073745660
edi          0xbffff010    -1073745904
eip          0x8048469      0x8048469 <comp+28>
eflags      0x286      [ PF SF IF ]
cs          0x73        115
ss          0x7b        123
ds          0x7b        123
es          0x7b        123
fs          0x0         0
gs          0x33        51
(gdb) bt
#0  0x08048469 in comp ()
#1  0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0
    ↪ 0xbffff0f8, n=n@entry=2)
    at msort.c:65
#2  0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0xbffff07c) at msort.c:
    ↪ .c:45
#3  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=
    ↪ =5) at msort.c:53
#4  0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0xbffff07c) at msort.c:
    ↪ .c:45
#5  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=
    ↪ =10) at msort.c:53
#6  0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0xbffff07c) at msort.c:
    ↪ 45
#7  __GI_qsorth_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry=4, cmp=
    ↪ cmp@entry=0x804844d <comp>,
    arg=arg@entry=0x0) at msort.c:297
#8  0xb7e42dcf in __GI_qsorth (b=0xbffff0f8, n=10, s=4, cmp=0x804844d <comp>
    ↪ >) at msort.c:307
#9  0x0804850d in main ()

```

### 1.33.3. Опасность указателей на ф-ции

Как мы можем видеть, ф-ция `qsort()` ожидает указатель на ф-цию, которая берет на вход два аргумента типа `void*` и возвращает целочисленное число. Если в вашем коде есть несколько разных ф-ций сравнения (одна сравнивает строки, другая — числа, и т. д.), очень легко их перепутать друг с другом. Вы можете попытаться отсортировать массив строк используя ф-цию сравнивающую числа, и компилятор не предупредит вас об ошибке.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

## 1.34. 64-битные значения в 32-битной среде

В среде, где **GPR!**-ы 32-битные, 64-битные значения хранятся и передаются как пары 32-битных значений <sup>171</sup>.

### 1.34.1. Возврат 64-битного значения

```
#include <stdint.h>

uint64_t f ()
{
    return 0x1234567890ABCDEF;
};
```

#### x86

64-битные значения в 32-битной среде возвращаются из функций в паре регистров EDX:EAX.

Листинг 1.372: Оптимизирующий MSVC 2010

```
_f    PROC
      mov     eax, -1867788817 ; 90abcdefH
      mov     edx, 305419896   ; 12345678H
      ret     0
_f    ENDP
```

#### ARM

64-битное значение возвращается в паре регистров R0-R1 — (R1 это старшая часть и R0 — младшая часть):

Листинг 1.373: Оптимизирующий Keil 6/2013 (Режим ARM)

```
||f|| PROC
      LDR     r0, |L0.12|
      LDR     r1, |L0.16|
      BX     lr
      ENDP

|L0.12|
      DCD     0x90abcdef

|L0.16|
      DCD     0x12345678
```

<sup>171</sup>Кстати, в 16-битной среде, 32-битные значения передаются 16-битными парами точно так же: ?? (стр. ??)

**MIPS**

64-битное значение возвращается в паре регистров V0-V1 (\$2-\$3) — (V0 (\$2) это старшая часть и V1 (\$3) — младшая часть):

Листинг 1.374: Оптимизирующий GCC 4.4.5 (assembly listing)

```
li    $3,-1867841536 # 0xffffffff90ab0000
li    $2,305397760   # 0x12340000
ori   $3,$3,0xcdef
j     $31
ori   $2,$2,0x5678
```

Листинг 1.375: Оптимизирующий GCC 4.4.5 (IDA)

```
lui   $v1, 0x90AB
lui   $v0, 0x1234
li    $v1, 0x90ABCDEF
jr    $ra
li    $v0, 0x12345678
```

**1.34.2. Передача аргументов, сложение, вычитание**

```
#include <stdint.h>

uint64_t f_add (uint64_t a, uint64_t b)
{
    return a+b;
};

void f_add_test ()
{
#ifdef __GNUC__
    printf ("%lld\n", f_add(12345678901234, 2345678901234));
#else
    printf ("%I64d\n", f_add(12345678901234, 2345678901234));
#endif
};

uint64_t f_sub (uint64_t a, uint64_t b)
{
    return a-b;
};
```

**x86**

Листинг 1.376: Оптимизирующий MSVC 2012 /Ob1

```
_a$ = 8      ; size = 8
_b$ = 16     ; size = 8
_f_add PROC
    mov     eax, DWORD PTR _a$[esp-4]
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

    add    eax, DWORD PTR _b$[esp-4]
    mov    edx, DWORD PTR _a$[esp]
    adc    edx, DWORD PTR _b$[esp]
    ret    0
_f_add   ENDP

_f_add_test PROC
    push  5461             ; 00001555H
    push  1972608889      ; 75939f79H
    push  2874            ; 00000b3aH
    push  1942892530      ; 73ce2ff2H
    call  _f_add
    push  edx
    push  eax
    push  OFFSET $SG1436 ; '%I64d', 0aH, 00H
    call  _printf
    add   esp, 28
    ret   0
_f_add_test ENDP

_f_sub   PROC
    mov    eax, DWORD PTR _a$[esp-4]
    sub    eax, DWORD PTR _b$[esp-4]
    mov    edx, DWORD PTR _a$[esp]
    sbb   edx, DWORD PTR _b$[esp]
    ret    0
_f_sub   ENDP

```

В `f_add_test()` видно, как каждое 64-битное число передается двумя 32-битными значениями, сначала старшая часть, затем младшая.

Сложение и вычитание происходит также парами.

При сложении, в начале складываются младшие 32 бита. Если при сложении был перенос, выставляется флаг CF. Следующая инструкция ADC складывает старшие части чисел, но также прибавляет единицу если  $CF = 1$ .

Вычитание также происходит парами. Первый SUB может также включить флаг переноса CF, который затем будет проверяться в SBB: если флаг переноса включен, то от результата отнимется единица.

Легко увидеть, как результат работы `f_add()` затем передается в `printf()`.

Листинг 1.377: GCC 4.8.1 -O1 -fno-inline

```

_f_add:
    mov    eax, DWORD PTR [esp+12]
    mov    edx, DWORD PTR [esp+16]
    add    eax, DWORD PTR [esp+4]
    adc    edx, DWORD PTR [esp+8]
    ret

_f_add_test:
    sub    esp, 28
    mov    DWORD PTR [esp+8], 1972608889 ; 75939f79H

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov     DWORD PTR [esp+12], 5461          ; 00001555H
mov     DWORD PTR [esp], 1942892530     ; 73ce2ff2H
mov     DWORD PTR [esp+4], 2874         ; 00000b3aH
call    _f_add
mov     DWORD PTR [esp+4], eax
mov     DWORD PTR [esp+8], edx
mov     DWORD PTR [esp], OFFSET FLAT:LC0 ; "%lld\n"
call    _printf
add     esp, 28
ret

_f_sub:
mov     eax, DWORD PTR [esp+4]
mov     edx, DWORD PTR [esp+8]
sub     eax, DWORD PTR [esp+12]
sbb     edx, DWORD PTR [esp+16]
ret

```

Код GCC почти такой же.

## ARM

Листинг 1.378: Оптимизирующий Keil 6/2013 (Режим ARM)

```

f_add PROC
    ADDS    r0,r0,r2
    ADC     r1,r1,r3
    BX      lr
    ENDP

f_sub PROC
    SUBS    r0,r0,r2
    SBC     r1,r1,r3
    BX      lr
    ENDP

f_add_test PROC
    PUSH    {r4,lr}
    LDR     r2,|L0.68| ; 0x75939f79
    LDR     r3,|L0.72| ; 0x00001555
    LDR     r0,|L0.76| ; 0x73ce2ff2
    LDR     r1,|L0.80| ; 0x00000b3a
    BL     f_add
    POP     {r4,lr}
    MOV     r2,r0
    MOV     r3,r1
    ADR     r0,|L0.84| ; "%I64d\n"
    B       __2printf
    ENDP

|L0.68|
    DCD     0x75939f79
|L0.72|

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

	DCD	0x00001555
L0.76	DCD	0x73ce2ff2
L0.80	DCD	0x00000b3a
L0.84	DCB	"%I64d\n",0

Первое 64-битное значение передается в паре регистров R0 и R1, второе — в паре R2 и R3. В ARM также есть инструкция ADC (учитывающая флаг переноса) и SBC («subtract with carry» — вычесть с переносом). Важная вещь: когда младшие части слагаются/вычитаются, используются инструкции ADDS и SUBS с суффиксом -S. Суффикс -S означает «set flags» (установить флаги), а флаги (особенно флаг переноса) это то что однозначно нужно последующим инструкциями ADC/SBC. А иначе инструкции без суффикса -S здесь вполне бы подошли (ADD и SUB).

## MIPS

Листинг 1.379: Оптимизирующий GCC 4.4.5 (IDA)

```
f_add:
; $a0 - старшая часть a
; $a1 - младшая часть a
; $a2 - старшая часть b
; $a3 - младшая часть b
        addu    $v1, $a3, $a1 ; суммировать младшие части
        addu    $a0, $a2, $a0 ; суммировать старшие части
; будет ли перенос сгенерирован во время суммирования младших частей?
; установить $v0 в 1, если да
        sltu    $v0, $v1, $a3
        jr      $ra
; прибавить 1 к старшей части результата, если перенос должен был быть
сгенерирован
        addu    $v0, $a0 ; branch delay slot
; $v0 - старшая часть результата
; $v1 - младшая часть результата

f_sub:
; $a0 - старшая часть a
; $a1 - младшая часть a
; $a2 - старшая часть b
; $a3 - младшая часть b
        subu    $v1, $a1, $a3 ; вычитать младшие части
        subu    $v0, $a0, $a2 ; вычитать старшие части
; будет ли перенос сгенерирован во время вычитания младших частей?
; установить $a0 в 1, если да
        sltu    $a1, $v1
        jr      $ra
; вычесть 1 из старшей части результата, если перенос должен был быть
сгенерирован
        subu    $v0, $a1 ; branch delay slot
; $v0 - старшая часть результата
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; $v1 - младшая часть результата

f_add_test:

var_10      = -0x10
var_4       = -4

        lui    $gp, (__gnu_local_gp >> 16)
        addiu  $sp, -0x20
        la    $gp, (__gnu_local_gp & 0xFFFF)
        sw    $ra, 0x20+var_4($sp)
        sw    $gp, 0x20+var_10($sp)
        lui   $a1, 0x73CE
        lui   $a3, 0x7593
        li    $a0, 0xB3A
        li    $a3, 0x75939F79
        li    $a2, 0x1555
        jal   f_add
        li    $a1, 0x73CE2FF2
        lw    $gp, 0x20+var_10($sp)
        lui   $a0, ($LC0 >> 16) # "%lld\n"
        lw    $t9, (printf & 0xFFFF)($gp)
        lw    $ra, 0x20+var_4($sp)
        la    $a0, ($LC0 & 0xFFFF) # "%lld\n"
        move  $a3, $v1
        move  $a2, $v0
        jr    $t9
        addiu $sp, 0x20

$LC0:    .ascii "%lld\n"<0>

```

В MIPS нет регистра флагов, так что эта информация не присутствует после исполнения арифметических операций.

Так что здесь нет инструкций как ADC или SBB в x86. Чтобы получить информацию о том, был бы выставлен флаг переноса, происходит сравнение (используя инструкцию SLTU), которая выставляет целевой регистр в 1 или 0.

Эта 1 или 0 затем прибавляется к итоговому результату, или вычитается.

### 1.34.3. Умножение, деление

```

#include <stdint.h>

uint64_t f_mul (uint64_t a, uint64_t b)
{
    return a*b;
};

uint64_t f_div (uint64_t a, uint64_t b)
{
    return a/b;
};

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
uint64_t f_rem (uint64_t a, uint64_t b)
{
    return a % b;
};
```

**x86**

Листинг 1.380: Оптимизирующий MSVC 2013 /Ob1

```
_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_mul PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _b$[ebp+4]
    push   eax
    mov     ecx, DWORD PTR _b$[ebp]
    push   ecx
    mov     edx, DWORD PTR _a$[ebp+4]
    push   edx
    mov     eax, DWORD PTR _a$[ebp]
    push   eax
    call   __allmul ; long long multiplication (умножение значений типа
long long)
    pop     ebp
    ret     0
_f_mul ENDP

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_div PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _b$[ebp+4]
    push   eax
    mov     ecx, DWORD PTR _b$[ebp]
    push   ecx
    mov     edx, DWORD PTR _a$[ebp+4]
    push   edx
    mov     eax, DWORD PTR _a$[ebp]
    push   eax
    call   __aulldiv ; unsigned long long division (деление беззнаковых
значений типа long long)
    pop     ebp
    ret     0
_f_div ENDP

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_rem PROC
    push    ebp
    mov     ebp, esp
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov     eax, DWORD PTR _b$[ebp+4]
push   eax
mov     ecx, DWORD PTR _b$[ebp]
push   ecx
mov     edx, DWORD PTR _a$[ebp+4]
push   edx
mov     eax, DWORD PTR _a$[ebp]
push   eax
call   __aullrem ; unsigned long long remainder (вычисление
беззнакового остатка)
pop     ebp
ret     0
_f_rem ENDP

```

Умножение и деление — это более сложная операция, так что обычно, компилятор встраивает вызовы библиотечных функций, делающих это.

Значение этих библиотечных функций, здесь: ?? (стр. ??).

Листинг 1.381: Оптимизирующий GCC 4.8.1 -fno-inline

```

_f_mul:
push   ebx
mov     edx, DWORD PTR [esp+8]
mov     eax, DWORD PTR [esp+16]
mov     ebx, DWORD PTR [esp+12]
mov     ecx, DWORD PTR [esp+20]
imul   ebx, eax
imul   ecx, edx
mul     edx
add     ecx, ebx
add     edx, ecx
pop     ebx
ret

_f_div:
sub     esp, 28
mov     eax, DWORD PTR [esp+40]
mov     edx, DWORD PTR [esp+44]
mov     DWORD PTR [esp+8], eax
mov     eax, DWORD PTR [esp+32]
mov     DWORD PTR [esp+12], edx
mov     edx, DWORD PTR [esp+36]
mov     DWORD PTR [esp], eax
mov     DWORD PTR [esp+4], edx
call   __udivdi3 ; unsigned division (беззнаковое деление)
add     esp, 28
ret

_f_rem:
sub     esp, 28
mov     eax, DWORD PTR [esp+40]
mov     edx, DWORD PTR [esp+44]
mov     DWORD PTR [esp+8], eax
mov     eax, DWORD PTR [esp+32]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov     DWORD PTR [esp+12], edx
mov     edx, DWORD PTR [esp+36]
mov     DWORD PTR [esp], eax
mov     DWORD PTR [esp+4], edx
call   ___umoddi3 ; unsigned modulo (беззнаковый остаток)
add     esp, 28
ret

```

GCC делает почти то же самое, тем не менее, встраивает код умножения прямо в функцию, посчитав что так будет эффективнее. У GCC другие имена библиотечных функций: ?? (стр. ??).

## ARM

Keil для режима Thumb вставляет вызовы библиотечных функций:

Листинг 1.382: Оптимизирующий Keil 6/2013 (Режим Thumb)

```

||f_mul|| PROC
PUSH    {r4,lr}
BL      __aeabi_lmul
POP     {r4,pc}
ENDP

||f_div|| PROC
PUSH    {r4,lr}
BL      __aeabi_udivmod
POP     {r4,pc}
ENDP

||f_rem|| PROC
PUSH    {r4,lr}
BL      __aeabi_udivmod
MOVS    r0,r2
MOVS    r1,r3
POP     {r4,pc}
ENDP

```

Keil для режима ARM, тем не менее, может сгенерировать код для умножения 64-битных чисел:

Листинг 1.383: Оптимизирующий Keil 6/2013 (Режим ARM)

```

||f_mul|| PROC
PUSH    {r4,lr}
UMULL   r12,r4,r0,r2
MLA     r1,r2,r1,r4
MLA     r1,r0,r3,r1
MOV     r0,r12
POP     {r4,pc}
ENDP

||f_div|| PROC

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    PUSH    {r4,lr}
    BL      __aeabi_uldivmod
    POP     {r4,pc}
    ENDP

||f_rem|| PROC
    PUSH    {r4,lr}
    BL      __aeabi_uldivmod
    MOV     r0,r2
    MOV     r1,r3
    POP     {r4,pc}
    ENDP

```

## MIPS

Оптимизирующий GCC для MIPS может генерировать код для 64-битного умножения, но для 64-битного деления приходится вызывать библиотечную функцию:

Листинг 1.384: Оптимизирующий GCC 4.4.5 (IDA)

```

f_mul:
    mult    $a2, $a1
    mflo    $v0
    or      $at, $zero ; NOP
    or      $at, $zero ; NOP
    mult    $a0, $a3
    mflo    $a0
    addu    $v0, $a0
    or      $at, $zero ; NOP
    multu   $a3, $a1
    mfhi    $a2
    mflo    $v1
    jr      $ra
    addu    $v0, $a2

f_div:
var_10 = -0x10
var_4 = -4

    lui    $gp, (__gnu_local_gp >> 16)
    addiu  $sp, -0x20
    la     $gp, (__gnu_local_gp & 0xFFFF)
    sw     $ra, 0x20+var_4($sp)
    sw     $gp, 0x20+var_10($sp)
    lw     $t9, (__udivdi3 & 0xFFFF)($gp)
    or     $at, $zero
    jalr   $t9
    or     $at, $zero
    lw     $ra, 0x20+var_4($sp)
    or     $at, $zero
    jr     $ra

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

        addiu    $sp, 0x20

f_rem:
var_10 = -0x10
var_4  = -4

        lui     $gp, (__gnu_local_gp >> 16)
        addiu  $sp, -0x20
        la     $gp, (__gnu_local_gp & 0xFFFF)
        sw     $ra, 0x20+var_4($sp)
        sw     $gp, 0x20+var_10($sp)
        lw     $t9, (__umoddi3 & 0xFFFF)($gp)
        or     $at, $zero
        jalr   $t9
        or     $at, $zero
        lw     $ra, 0x20+var_4($sp)
        or     $at, $zero
        jr    $ra
        addiu  $sp, 0x20

```

Тут также много **NOP!**-ов, это возможно заполнение delay slot-ов после инструкции умножения (она ведь работает медленнее прочих инструкций).

#### 1.34.4. Сдвиг вправо

```

#include <stdint.h>

uint64_t f (uint64_t a)
{
    return a>>7;
};

```

#### x86

Листинг 1.385: Оптимизирующий MSVC 2012 /Ob1

```

_a$ = 8          ; size = 8
_f PROC
    mov     eax, DWORD PTR _a$[esp-4]
    mov     edx, DWORD PTR _a$[esp]
    shrd   eax, edx, 7
    shr    edx, 7
    ret    0
_f ENDP

```

Листинг 1.386: Оптимизирующий GCC 4.8.1 -fno-inline

```

_f:
    mov     edx, DWORD PTR [esp+8]
    mov     eax, DWORD PTR [esp+4]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
shrd    eax, edx, 7
shr     edx, 7
ret
```

Сдвиг происходит также в две операции: в начале сдвигается младшая часть, затем старшая. Но младшая часть сдвигается при помощи инструкции SHRD, она сдвигает значение в EAX на 7 бит, но подтягивает новые биты из EDX, т.е. из старшей части. Другими словами, 64-битное значение из пары регистров EDX:EAX, как одно целое, сдвигается на 7 бит и младшие 32 бита результата сохраняются в EAX. Старшая часть сдвигается куда более популярной инструкцией SHR: действительно, ведь освободившиеся биты в старшей части нужно просто заполнить нулями.

## ARM

В ARM нет такой инструкции как SHRD в x86, так что компилятору Keil приходится всё это делать, используя простые сдвиги и операции «ИЛИ»:

Листинг 1.387: Оптимизирующий Keil 6/2013 (Режим ARM)

```
||f|| PROC
    LSR    r0, r0, #7
    ORR    r0, r0, r1, LSL #25
    LSR    r1, r1, #7
    BX     lr
ENDP
```

Листинг 1.388: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
||f|| PROC
    LSLS   r2, r1, #25
    LSRS   r0, r0, #7
    ORRS   r0, r0, r2
    LSRS   r1, r1, #7
    BX     lr
ENDP
```

## MIPS

GCC для MIPS реализует тот же алгоритм, что сделал Keil для режима Thumb:

Листинг 1.389: Оптимизирующий GCC 4.4.5 (IDA)

```
f:
    sll    $v0, $a0, 25
    srl    $v1, $a1, 7
    or     $v1, $v0, $v1
    jr     $ra
    srl    $v0, $a0, 7
```

### 1.34.5. Конвертирование 32-битного значения в 64-битное

```
#include <stdint.h>

int64_t f (int32_t a)
{
    return a;
};
```

#### x86

Листинг 1.390: Оптимизирующий MSVC 2012

```
_a$ = 8
_f    PROC
      mov     eax, DWORD PTR _a$[esp-4]
      cdq
      ret     0
_f    ENDP
```

Здесь появляется необходимость расширить 32-битное знаковое значение в 64-битное знаковое.

Конвертировать беззнаковые значения очень просто: нужно просто выставить в 0 все биты в старшей части. Но для знаковых типов это не подходит: знак числа должен быть скопирован в старшую часть числа-результата. Здесь это делает инструкция CDQ, она берет входное значение в EAX, расширяет его до 64-битного, и оставляет его в паре регистров EDX:EAX. Иными словами, инструкция CDQ узнает знак числа в EAX (просто берет самый старший бит в EAX) и в зависимости от этого, выставляет все 32 бита в EDX в 0 или в 1. Её работа в каком-то смысле напоминает работу инструкции MOVSX.

#### ARM

Листинг 1.391: Оптимизирующий Keil 6/2013 (Режим ARM)

```
||f|| PROC
      ASR     r1,r0,#31
      BX     lr
      ENDP
```

Keil для ARM работает иначе: он просто сдвигает (арифметически) входное значение на 31 бит вправо. Как мы знаем, бит знака это **MSB!**, и арифметический сдвиг копирует бит знака в «появляющихся» битах.

Так что после инструкции ASR r1,r0,#31, R1 будет содержать 0xFFFFFFFF если входное значение было отрицательным, или 0 в противном случае. R1 содержит старшую часть возвращаемого 64-битного значения. Другими словами, этот код просто копирует **MSB!** (бит знака) из входного значения в R0 во все биты старшей 32-битной части итогового 64-битного значения.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

---

**MIPS**

GCC для MIPS делает то же, что сделал Keil для режима ARM:

Листинг 1.392: Оптимизирующий GCC 4.4.5 (IDA)

```
f:
  sra    $v0, $a0, 31
  jr     $ra
  move   $v1, $a0
```

## 1.35. Случай со структурой `LARGE_INTEGER`

Представьте: конец 1980-х, вы Microsoft, и вы разрабатываете новую *серьезную OS!* (Windows NT), которая будет конкурировать с Юниксами. Целевые платформы это и 32-битные и 64-битные процессоры. И вам нужен 64-битный целочисленный тип для самых разных целей, начиная со структуры FILETIME<sup>172</sup>.

Проблема: пока еще не все компиляторы с Си/Си++ поддерживают 64-битные целочисленные (это конец 1980-х). Конечно, это изменится в (ближайшем) будущем, но не сейчас. Что вы будете делать?

Во время чтения, попробуйте остановиться (и/или закрыть эту книгу) и подумать, как вы можете решить эту проблему.

---

<sup>172</sup><https://docs.microsoft.com/en-us/windows/desktop/api/minwinbase/ns-minwinbase-filetime>

И вот что сделали в Microsoft, что-то вроде этого<sup>173</sup>:

```
union ULARGE_INTEGER
{
    struct backward_compatibility
    {
        DWORD LowPart;
        DWORD HighPart;
    };
#ifdef NEW_FANCY_COMPILER_SUPPORTING_64_BIT
    ULONGLONG QuadPart;
#endif
};
```

Это кусок из 8-и байт, к которому можно обратиться через 64-битное целочисленное QuadPart (если скомпилировано новым компилятором), или используя два 32-битных целочисленных (если скомпилировано старым компилятором).

Поле QuadPart просто отсутствует здесь, если компилируется старым компилятором.

Порядок существенен: первое поле (LowPart) транслируется в младшие 4 байта 64-битного значения, второе поле (HighPart) — в старшие 4 байта.

В Microsoft также добавили ф-ции для арифметических операций, в той же манере, что уже было описано ранее: ?? (стр. ??).

Вот что можно найти в утекших исходных файлах Windows 2000:

#### Листинг 1.393: Архитектура i386

```
;++
;
; LARGE_INTEGER
; RtlLargeIntegerAdd (
; IN LARGE_INTEGER Addend1,
; IN LARGE_INTEGER Addend2
; )
;
; Routine Description:
;
; This function adds a signed large integer to a signed large integer and
; returns the signed large integer result.
;
; Arguments:
;
; (TOS+4) = Addend1 - first addend value
; (TOS+12) = Addend2 - second addend value
;
; Return Value:
;
; The large integer result is stored in (edx:eax)
;
;--
```

<sup>173</sup>Это не копипаста, я сам это написал

```

cPublicProc _RtlLargeIntegerAdd ,4
cPublicFpo 4,0

    mov     eax,[esp]+4           ; (eax)=add1.low
    add     eax,[esp]+12         ; (eax)=sum.low
    mov     edx,[esp]+8           ; (edx)=add1.hi
    adc     edx,[esp]+16         ; (edx)=sum.hi
    stdRET  _RtlLargeIntegerAdd

stdENDP _RtlLargeIntegerAdd

```

#### Листинг 1.394: Архитектура MIPS

```

LEAF_ENTRY(RtlLargeIntegerAdd)

    lw     t0,4 * 4(sp)         // get low part of addend2 value
    lw     t1,4 * 5(sp)         // get high part of addend2 value
    addu   t0,t0,a2             // add low parts of large integer
    addu   t1,t1,a3             // add high parts of large integer
    sltu   t2,t0,a2             // generate carry from low part
    addu   t1,t1,t2             // add carry to high part
    sw     t0,0(a0)             // store low part of result
    sw     t1,4(a0)             // store high part of result
    move   v0,a0                // set function return register
    j      ra                    // return

.end    RtlLargeIntegerAdd

```

Теперь две 64-битных архитектуры:

#### Листинг 1.395: Архитектура Itanium

```

LEAF_ENTRY(RtlLargeIntegerAdd)

    add     v0 = a0, a1          // add both quadword ↙
↘ arguments
    LEAF_RETURN

LEAF_EXIT(RtlLargeIntegerAdd)

```

#### Листинг 1.396: Архитектура DEC Alpha

```

LEAF_ENTRY(RtlLargeIntegerAdd)

    addq   a0, a1, v0           // add both quadword arguments
    ret    zero, (ra)           // return

.end    RtlLargeIntegerAdd

```

В Itanium и DEC Alpha не нужно использовать 32-битные инструкции, потому что уже есть 64-битные.

И вот что можно найти в Windows Research Kernel:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

DECLSPEC_DEPRECATED_DDK          // Use native __int64 math
__inline
LARGE_INTEGER
NTAPI
RtlLargeIntegerAdd (
    LARGE_INTEGER Addend1,
    LARGE_INTEGER Addend2
)
{
    LARGE_INTEGER Sum;

    Sum.QuadPart = Addend1.QuadPart + Addend2.QuadPart;
    return Sum;
}

```

Все эти ф-ции могут быть убраны (в будущем), но сейчас они работают с полем QuadPart. Если этот фрагмент кода будет скомпилирован современным 32-битным компилятором (поддерживающим 64-битные целочисленные), он сгенерирует два 32-битных сложения. С этого момента, поля LowPart/HighPart можно убрать из структуры LARGE\_INTEGER.

Нужно ли использовать такую технику сегодня? Вероятно нет, но если кому-то вдруг понадобится 128-битный тип целочисленных, вы можете его сделать точно так же.

Так же, нужно сказать, это работает благодаря порядку байт *little-endian* (?? (стр. ??)) (все архитектуры под которые разрабатывалась Windows NT, *little-endian*). Этот трюк не сработает на *big-endian*-архитектуре.

## 1.36. SIMD

**SIMD!** это акроним: *Single Instruction, Multiple Data*.

Как можно судить по названию, это обработка множества данных исполняя только одну инструкцию.

Как и **FPU!**, эта подсистема процессора выглядит так же отдельным процессором внутри x86.

SIMD в x86 начался с MMX. Появилось 8 64-битных регистров MM0-MM7.

Каждый MMX-регистр может содержать 2 32-битных значения, 4 16-битных или же 8 байт. Например, складывая значения двух MMX-регистров, можно складывать одновременно 8 8-битных значений.

Простой пример, это некий графический редактор, который хранит открытое изображение как двумерный массив. Когда пользователь меняет яркость изображения, редактору нужно, например, прибавить некий коэффициент ко всем пикселям, или отнять. Для простоты можно представить, что изображение у нас бело-серо-черное и каждый пиксель занимает один байт, то с помощью MMX можно менять яркость сразу у восьми пикселей.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Кстати, вот причина почему в SIMD присутствуют инструкции с *насыщением* (*saturation*).

Когда пользователь в графическом редакторе изменяет яркость, переполнение и антипереполнение (*underflow*) не нужны, так что в SIMD имеются, например, инструкции сложения, которые ничего не будут прибавлять если максимальное значение уже достигнуто, и т. д.

Когда MMX только появилось, эти регистры на самом деле располагались в FPU-регистрах. Можно было использовать либо FPU либо MMX в одно и то же время. Можно подумать, что Intel решило немного сэкономить на транзисторах, но на самом деле причина такого симбиоза проще — более старая **OS!** не знающая о дополнительных регистрах процессора не будет сохранять их во время переключения задач, а вот регистры FPU сохранять будет. Таким образом, процессор с MMX + старая **OS!** + задача, использующая возможности MMX = все это может работать вместе.

SSE — это расширение регистров до 128 бит, теперь уже отдельно от FPU.

AVX — расширение регистров до 256 бит.

Немного о практическом применении.

Конечно же, это копирование блоков в памяти (*memcpr*), сравнение (*memcpr*), и подобное.

Еще пример: имеется алгоритм шифрования DES, который берет 64-битный блок, 56-битный ключ, шифрует блок с ключом и образуется 64-битный результат. Алгоритм DES можно легко представить в виде очень большой электронной цифровой схемы, с проводами, элементами И, ИЛИ, НЕ.

Идея *bitslice* DES<sup>174</sup> — это обработка сразу группы блоков и ключей одновременно. Скажем, на x86 переменная типа *unsigned int* вмещает в себе 32 бита, так что там можно хранить промежуточные результаты сразу для 32-х блоков-ключей, используя 64+56 переменных типа *unsigned int*.

Существует утилита для перебора паролей/хешей Oracle RDBMS (которые основаны на алгоритме DES), реализующая алгоритм *bitslice* DES для SSE2 и AVX — и теперь возможно шифровать одновременно 128 или 256 блоков-ключей:

[http://conus.info/utills/ops\\_SIMD/](http://conus.info/utills/ops_SIMD/)

### 1.36.1. Векторизация

Векторизация<sup>175</sup> это когда у вас есть цикл, который берет на вход несколько массивов и выдает, например, один массив данных. Тело цикла берет некоторые элементы из входных массивов, что-то делает с ними и помещает в выходной. Векторизация — это обрабатывать несколько элементов одновременно.

Векторизация — это не самая новая технология: автор сих строк видел её по крайней мере на линейке суперкомпьютеров Cray Y-MP от 1988, когда работал

<sup>174</sup><http://www.darkside.com.au/bitslice/>

<sup>175</sup>[Wikipedia: vectorization](#)



на его версии-«лайт» Cray Y-MP EL <sup>176</sup>.

Например:

```
for (i = 0; i < 1024; i++)
{
    C[i] = A[i]*B[i];
}
```

Этот фрагмент кода берет элементы из A и B, перемножает и сохраняет результат в C.

Если представить, что каждый элемент массива — это 32-битный *int*, то их можно загружать сразу по 4 из A в 128-битный XMM-регистр, из B в другой XMM-регистр и выполнив инструкцию *PMULLD* (*Перемножить упакованные знаковые DWORD и сохранить младшую часть результата*) и *PMULHW* (*Перемножить упакованные знаковые DWORD и сохранить старшую часть результата*), можно получить 4 64-битных **произведения** сразу.

Таким образом, тело цикла исполняется 1024/4 раза вместо 1024, что в 4 раза меньше, и, конечно, быстрее.

### Пример сложения

Некоторые компиляторы умеют делать автоматическую векторизацию в простых случаях, например, Intel C++ <sup>177</sup>.

Вот очень простая функция:

```
int f (int sz, int *ar1, int *ar2, int *ar3)
{
    for (int i=0; i<sz; i++)
        ar3[i]=ar1[i]+ar2[i];

    return 0;
};
```

### Intel C++

Компилируем её при помощи Intel C++ 11.1.051 win32:

```
icl intel.cpp /QaxSSE2 /Faintel.asm /Ox
```

Имеем такое (в **IDA!**):

```
; int __cdecl f(int, int *, int *, int *)
                public ?f@YАННРАН00@Z
?f@YАННРАН00@Z proc near
```

<sup>176</sup>Удаленно. Он находится в музее суперкомпьютеров: <http://www.cray-cyber.org>

<sup>177</sup>Еще о том, как Intel C++ умеет автоматически векторизовать циклы: [Excerpt: Effective Automatic Vectorization](#)

```

var_10 = dword ptr -10h
sz      = dword ptr 4
ar1     = dword ptr 8
ar2     = dword ptr 0Ch
ar3     = dword ptr 10h

    push    edi
    push    esi
    push    ebx
    push    esi
    mov     edx, [esp+10h+sz]
    test    edx, edx
    jle     loc_15B
    mov     eax, [esp+10h+ar3]
    cmp     edx, 6
    jle     loc_143
    cmp     eax, [esp+10h+ar2]
    jbe     short loc_36
    mov     esi, [esp+10h+ar2]
    sub     esi, eax
    lea    ecx, ds:0[edx*4]
    neg     esi
    cmp     ecx, esi
    jbe     short loc_55

loc_36: ; CODE XREF: f(int,int *,int *,int *)+21
    cmp     eax, [esp+10h+ar2]
    jnb     loc_143
    mov     esi, [esp+10h+ar2]
    sub     esi, eax
    lea    ecx, ds:0[edx*4]
    cmp     esi, ecx
    jb     loc_143

loc_55: ; CODE XREF: f(int,int *,int *,int *)+34
    cmp     eax, [esp+10h+ar1]
    jbe     short loc_67
    mov     esi, [esp+10h+ar1]
    sub     esi, eax
    neg     esi
    cmp     ecx, esi
    jbe     short loc_7F

loc_67: ; CODE XREF: f(int,int *,int *,int *)+59
    cmp     eax, [esp+10h+ar1]
    jnb     loc_143
    mov     esi, [esp+10h+ar1]
    sub     esi, eax
    cmp     esi, ecx
    jb     loc_143

loc_7F: ; CODE XREF: f(int,int *,int *,int *)+65
    mov     edi, eax          ; edi = ar3

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    and     edi, 0Fh      ; ar3 выровнен по 16-байтной границе?
    jz      short loc_9A ; да
    test    edi, 3
    jnz     loc_162
    neg     edi
    add     edi, 10h
    shr     edi, 2

loc_9A: ; CODE XREF: f(int,int *,int *,int *)+84
    lea     ecx, [edi+4]
    cmp     edx, ecx
    jl      loc_162
    mov     ecx, edx
    sub     ecx, edi
    and     ecx, 3
    neg     ecx
    add     ecx, edx
    test    edi, edi
    jbe     short loc_D6
    mov     ebx, [esp+10h+ar2]
    mov     [esp+10h+var_10], ecx
    mov     ecx, [esp+10h+ar1]
    xor     esi, esi

loc_C1: ; CODE XREF: f(int,int *,int *,int *)+CD
    mov     edx, [ecx+esi*4]
    add     edx, [ebx+esi*4]
    mov     [eax+esi*4], edx
    inc     esi
    cmp     esi, edi
    jb      short loc_C1
    mov     ecx, [esp+10h+var_10]
    mov     edx, [esp+10h+sz]

loc_D6: ; CODE XREF: f(int,int *,int *,int *)+B2
    mov     esi, [esp+10h+ar2]
    lea     esi, [esi+edi*4] ; ar2+i*4 выровнен по 16-байтной границе?
    test    esi, 0Fh
    jz      short loc_109 ; да!
    mov     ebx, [esp+10h+ar1]
    mov     esi, [esp+10h+ar2]

loc_ED: ; CODE XREF: f(int,int *,int *,int *)+105
    movdqu xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4
    movdqu xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 не выровнен по
16-байтной границе, так что загружаем это в XMM0
    padd    xmm1, xmm0
    movdqa xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4
    add     edi, 4
    cmp     edi, ecx
    jb      short loc_ED
    jmp     short loc_127

loc_109: ; CODE XREF: f(int,int *,int *,int *)+E3

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
    mov     ebx, [esp+10h+ar1]
    mov     esi, [esp+10h+ar2]

loc_111: ; CODE XREF: f(int,int *,int *,int *)+125
    movdqu xmm0, xmmword ptr [ebx+edi*4]
    padd   xmm0, xmmword ptr [esi+edi*4]
    movdqa xmmword ptr [eax+edi*4], xmm0
    add    edi, 4
    cmp    edi, ecx
    jb     short loc_111

loc_127: ; CODE XREF: f(int,int *,int *,int *)+107
        ; f(int,int *,int *,int *)+164
    cmp    ecx, edx
    jnb    short loc_15B
    mov    esi, [esp+10h+ar1]
    mov    edi, [esp+10h+ar2]

loc_133: ; CODE XREF: f(int,int *,int *,int *)+13F
    mov    ebx, [esi+ecx*4]
    add    ebx, [edi+ecx*4]
    mov    [eax+ecx*4], ebx
    inc    ecx
    cmp    ecx, edx
    jb     short loc_133
    jmp    short loc_15B

loc_143: ; CODE XREF: f(int,int *,int *,int *)+17
        ; f(int,int *,int *,int *)+3A ...
    mov    esi, [esp+10h+ar1]
    mov    edi, [esp+10h+ar2]
    xor    ecx, ecx

loc_14D: ; CODE XREF: f(int,int *,int *,int *)+159
    mov    ebx, [esi+ecx*4]
    add    ebx, [edi+ecx*4]
    mov    [eax+ecx*4], ebx
    inc    ecx
    cmp    ecx, edx
    jb     short loc_14D

loc_15B: ; CODE XREF: f(int,int *,int *,int *)+A
        ; f(int,int *,int *,int *)+129 ...
    xor    eax, eax
    pop    ecx
    pop    ebx
    pop    esi
    pop    edi
    retn

loc_162: ; CODE XREF: f(int,int *,int *,int *)+8C
        ; f(int,int *,int *,int *)+9F
    xor    ecx, ecx
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    jmp     short loc_127
?f@YАННРАН00@Z endp

```

Инструкции, имеющие отношение к SSE2 это:

- **MOVDQU** (*Move Unaligned Double Quadword*) — она просто загружает 16 байт из памяти в XMM-регистр.
- **PADDQ** (*Add Packed Integers*) — складывает сразу 4 пары 32-битных чисел и оставляет в первом операнде результат. Кстати, если произойдет переполнение, то исключения не произойдет и никакие флаги не установятся, запишутся просто младшие 32 бита результата. Если один из операндов **PADDQ** — адрес значения в памяти, то требуется чтобы адрес был выровнен по 16-байтной границе. Если он не выровнен, произойдет исключение.
- **MOVDAQ** (*Move Aligned Double Quadword*) — тоже что и **MOVDQU**, только подразумевает что адрес в памяти выровнен по 16-байтной границе. Если он не выровнен, произойдет исключение. **MOVDAQ** работает быстрее чем **MOVDQU**, но требует вышеозначенного.

Итак, эти SSE2-инструкции исполняются только в том случае если еще осталось просуммировать 4 пары переменных типа *int* плюс если указатель *ar3* выровнен по 16-байтной границе.

Более того, если еще и *ar2* выровнен по 16-байтной границе, то будет выполняться этот фрагмент кода:

```

movdqu xmm0, xmmword ptr [ebx+edi*4] ; ar1+i*4
paddq  xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4
movdqa xmmword ptr [eax+edi*4], xmm0 ; ar3+i*4

```

А иначе, значение из *ar2* загрузится в **XMM0** используя инструкцию **MOVDQU**, которая не требует выровненного указателя, зато может работать чуть медленнее:

```

movdqu xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4
movdqu xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 не выровнен по 16-байтной
    границе, так что загружаем это в XMM0
paddq  xmm1, xmm0
movdqa xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4

```

А во всех остальных случаях, будет исполняться код, который был бы, как если бы не была включена поддержка SSE2.

## GCC

Но и GCC умеет кое-что векторизировать<sup>178</sup>, если компилировать с опциями `-O3` и включить поддержку SSE2: `-msse2`.

Вот что вышло (GCC 4.4.1):

<sup>178</sup>Подробнее о векторизации в GCC: <http://gcc.gnu.org/projects/tree-ssa/vectorization.html>

```

; f(int, int *, int *, int *)
    public _Z1fiPiS_S_
_Z1fiPiS_S_ proc near

var_18    = dword ptr -18h
var_14    = dword ptr -14h
var_10    = dword ptr -10h
arg_0     = dword ptr  8
arg_4     = dword ptr  0Ch
arg_8     = dword ptr  10h
arg_C     = dword ptr  14h

        push    ebp
        mov     ebp, esp
        push    edi
        push    esi
        push    ebx
        sub     esp, 0Ch
        mov     ecx, [ebp+arg_0]
        mov     esi, [ebp+arg_4]
        mov     edi, [ebp+arg_8]
        mov     ebx, [ebp+arg_C]
        test    ecx, ecx
        jle     short loc_80484D8
        cmp     ecx, 6
        lea    eax, [ebx+10h]
        ja     short loc_80484E8

loc_80484C1: ; CODE XREF: f(int,int *,int *,int *)+4B
             ; f(int,int *,int *,int *)+61 ...
        xor     eax, eax
        nop
        lea    esi, [esi+0]

loc_80484C8: ; CODE XREF: f(int,int *,int *,int *)+36
        mov     edx, [edi+eax*4]
        add     edx, [esi+eax*4]
        mov     [ebx+eax*4], edx
        add     eax, 1
        cmp     eax, ecx
        jnz    short loc_80484C8

loc_80484D8: ; CODE XREF: f(int,int *,int *,int *)+17
             ; f(int,int *,int *,int *)+A5
        add     esp, 0Ch
        xor     eax, eax
        pop     ebx
        pop     esi
        pop     edi
        pop     ebp
        retn

        align 8

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
loc_80484E8: ; CODE XREF: f(int,int *,int *,int *)+1F
            test    bl, 0Fh
            jnz     short loc_80484C1
            lea    edx, [esi+10h]
            cmp    ebx, edx
            jbe    loc_8048578

loc_80484F8: ; CODE XREF: f(int,int *,int *,int *)+E0
            lea    edx, [edi+10h]
            cmp    ebx, edx
            ja     short loc_8048503
            cmp    edi, eax
            jbe    short loc_80484C1

loc_8048503: ; CODE XREF: f(int,int *,int *,int *)+5D
            mov    eax, ecx
            shr    eax, 2
            mov    [ebp+var_14], eax
            shl    eax, 2
            test   eax, eax
            mov    [ebp+var_10], eax
            jz     short loc_8048547
            mov    [ebp+var_18], ecx
            mov    ecx, [ebp+var_14]
            xor    eax, eax
            xor    edx, edx
            nop

loc_8048520: ; CODE XREF: f(int,int *,int *,int *)+9B
            movdqu xmm1, xmmword ptr [edi+eax]
            movdqu xmm0, xmmword ptr [esi+eax]
            add    edx, 1
            padd  xmm0, xmm1
            movdqa xmmword ptr [ebx+eax], xmm0
            add    eax, 10h
            cmp    edx, ecx
            jb     short loc_8048520
            mov    ecx, [ebp+var_18]
            mov    eax, [ebp+var_10]
            cmp    ecx, eax
            jz     short loc_80484D8

loc_8048547: ; CODE XREF: f(int,int *,int *,int *)+73
            lea    edx, ds:0[eax*4]
            add    esi, edx
            add    edi, edx
            add    ebx, edx
            lea    esi, [esi+0]

loc_8048558: ; CODE XREF: f(int,int *,int *,int *)+CC
            mov    edx, [edi]
            add    eax, 1
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    add     edi, 4
    add     edx, [esi]
    add     esi, 4
    mov     [ebx], edx
    add     ebx, 4
    cmp     ecx, eax
    jg      short loc_8048558
    add     esp, 0Ch
    xor     eax, eax
    pop     ebx
    pop     esi
    pop     edi
    pop     ebp
    retn

loc_8048578: ; CODE XREF: f(int,int *,int *,int *)+52
    cmp     eax, esi
    jnb     loc_80484C1
    jmp     loc_80484F8
_Z1fiPiS_S_ endp

```

Почти то же самое, хотя и не так дотошно, как Intel C++.

### Пример копирования блоков

Вернемся к простому примеру memcpy() (?? (стр. ??)):

```

#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
    size_t i;
    for (i=0; i<cnt; i++)
        dst[i]=src[i];
};

```

И вот что делает оптимизирующий GCC 4.9.1:

#### Листинг 1.397: Оптимизирующий GCC 4.9.1 x64

```

my_memcpy:
; RDI = адрес назначения
; RSI = исходный адрес
; RDX = размер блока
    test   rdx, rdx
    je     .L41
    lea   rax, [rdi+16]
    cmp   rsi, rax
    lea   rax, [rsi+16]
    setae cl
    cmp   rdi, rax
    setae al
    or    cl, al

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```
je .L13
cmp rdx, 22
jbe .L13
mov rcx, rsi
push rbp
push rbx
neg rcx
and ecx, 15
cmp rcx, rdx
cmova rcx, rdx
xor eax, eax
test rcx, rcx
je .L4
movzx eax, BYTE PTR [rsi]
cmp rcx, 1
mov BYTE PTR [rdi], al
je .L15
movzx eax, BYTE PTR [rsi+1]
cmp rcx, 2
mov BYTE PTR [rdi+1], al
je .L16
movzx eax, BYTE PTR [rsi+2]
cmp rcx, 3
mov BYTE PTR [rdi+2], al
je .L17
movzx eax, BYTE PTR [rsi+3]
cmp rcx, 4
mov BYTE PTR [rdi+3], al
je .L18
movzx eax, BYTE PTR [rsi+4]
cmp rcx, 5
mov BYTE PTR [rdi+4], al
je .L19
movzx eax, BYTE PTR [rsi+5]
cmp rcx, 6
mov BYTE PTR [rdi+5], al
je .L20
movzx eax, BYTE PTR [rsi+6]
cmp rcx, 7
mov BYTE PTR [rdi+6], al
je .L21
movzx eax, BYTE PTR [rsi+7]
cmp rcx, 8
mov BYTE PTR [rdi+7], al
je .L22
movzx eax, BYTE PTR [rsi+8]
cmp rcx, 9
mov BYTE PTR [rdi+8], al
je .L23
movzx eax, BYTE PTR [rsi+9]
cmp rcx, 10
mov BYTE PTR [rdi+9], al
je .L24
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мой адреса](#). Спасибо!

```

movzx  eax, BYTE PTR [rsi+10]
cmp    rcx, 11
mov    BYTE PTR [rdi+10], al
je     .L25
movzx  eax, BYTE PTR [rsi+11]
cmp    rcx, 12
mov    BYTE PTR [rdi+11], al
je     .L26
movzx  eax, BYTE PTR [rsi+12]
cmp    rcx, 13
mov    BYTE PTR [rdi+12], al
je     .L27
movzx  eax, BYTE PTR [rsi+13]
cmp    rcx, 15
mov    BYTE PTR [rdi+13], al
jne    .L28
movzx  eax, BYTE PTR [rsi+14]
mov    BYTE PTR [rdi+14], al
mov    eax, 15
.L4:
mov    r10, rdx
lea    r9, [rdx-1]
sub    r10, rcx
lea    r8, [r10-16]
sub    r9, rcx
shr    r8, 4
add    r8, 1
mov    r11, r8
sal    r11, 4
cmp    r9, 14
jbe    .L6
lea    rbp, [rsi+rcx]
xor    r9d, r9d
add    rcx, rdi
xor    ebx, ebx
.L7:
movdqa xmm0, XMMWORD PTR [rbp+0+r9]
add    rbx, 1
movups XMMWORD PTR [rcx+r9], xmm0
add    r9, 16
cmp    rbx, r8
jb     .L7
add    rax, r11
cmp    r10, r11
je     .L1
.L6:
movzx  ecx, BYTE PTR [rsi+rax]
mov    BYTE PTR [rdi+rax], cl
lea    rcx, [rax+1]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+1+rax]
mov    BYTE PTR [rdi+1+rax], cl

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
lea    rcx, [rax+2]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+2+rax]
mov    BYTE PTR [rdi+2+rax], cl
lea    rcx, [rax+3]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+3+rax]
mov    BYTE PTR [rdi+3+rax], cl
lea    rcx, [rax+4]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+4+rax]
mov    BYTE PTR [rdi+4+rax], cl
lea    rcx, [rax+5]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+5+rax]
mov    BYTE PTR [rdi+5+rax], cl
lea    rcx, [rax+6]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+6+rax]
mov    BYTE PTR [rdi+6+rax], cl
lea    rcx, [rax+7]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+7+rax]
mov    BYTE PTR [rdi+7+rax], cl
lea    rcx, [rax+8]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+8+rax]
mov    BYTE PTR [rdi+8+rax], cl
lea    rcx, [rax+9]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+9+rax]
mov    BYTE PTR [rdi+9+rax], cl
lea    rcx, [rax+10]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+10+rax]
mov    BYTE PTR [rdi+10+rax], cl
lea    rcx, [rax+11]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+11+rax]
mov    BYTE PTR [rdi+11+rax], cl
lea    rcx, [rax+12]
cmp    rdx, rcx
jbe    .L1
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
    movzx ecx, BYTE PTR [rsi+12+rax]
    mov    BYTE PTR [rdi+12+rax], cl
    lea   rcx, [rax+13]
    cmp   rdx, rcx
    jbe   .L1
    movzx ecx, BYTE PTR [rsi+13+rax]
    mov    BYTE PTR [rdi+13+rax], cl
    lea   rcx, [rax+14]
    cmp   rdx, rcx
    jbe   .L1
    movzx edx, BYTE PTR [rsi+14+rax]
    mov    BYTE PTR [rdi+14+rax], dl
.L1:
    pop   rbx
    pop   rbp
.L41:
    rep  ret
.L13:
    xor   eax, eax
.L3:
    movzx ecx, BYTE PTR [rsi+rax]
    mov    BYTE PTR [rdi+rax], cl
    add   rax, 1
    cmp   rax, rdx
    jne   .L3
    rep  ret
.L28:
    mov   eax, 14
    jmp   .L4
.L15:
    mov   eax, 1
    jmp   .L4
.L16:
    mov   eax, 2
    jmp   .L4
.L17:
    mov   eax, 3
    jmp   .L4
.L18:
    mov   eax, 4
    jmp   .L4
.L19:
    mov   eax, 5
    jmp   .L4
.L20:
    mov   eax, 6
    jmp   .L4
.L21:
    mov   eax, 7
    jmp   .L4
.L22:
    mov   eax, 8
    jmp   .L4
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.L23:
    mov    eax, 9
    jmp    .L4
.L24:
    mov    eax, 10
    jmp    .L4
.L25:
    mov    eax, 11
    jmp    .L4
.L26:
    mov    eax, 12
    jmp    .L4
.L27:
    mov    eax, 13
    jmp    .L4

```

### 1.36.2. Реализация `strlen()` при помощи SIMD

Прежде всего, следует заметить, что SIMD-инструкции можно вставлять в Си/Си++ код при помощи специальных макросов<sup>179</sup>. В MSVC, часть находится в файле `intrin.h`.

Имеется возможность реализовать функцию `strlen()`<sup>180</sup> при помощи SIMD-инструкций, работающий в 2-2.5 раза быстрее обычной реализации. Эта функция будет загружать в XMM-регистр сразу 16 байт и проверять каждый на ноль

<sup>181</sup>.

```

size_t strlen_sse2(const char *str)
{
    register size_t len = 0;
    const char *s=str;
    bool str_is_aligned=(((unsigned int)str)&0xFFFFFFFF0) == (unsigned int)↵
    ↵ str;

    if (str_is_aligned==false)
        return strlen (str);

    __m128i xmm0 = _mm_setzero_si128();
    __m128i xmm1;
    int mask = 0;

    for (;;)
    {
        xmm1 = _mm_load_si128((__m128i *)s);
        xmm1 = _mm_cmpeq_epi8(xmm1, xmm0);
        if ((mask = _mm_movemask_epi8(xmm1)) != 0)
        {
            unsigned long pos;

```

<sup>179</sup>MSDN: MMX, SSE, and SSE2 Intrinsics

<sup>180</sup>`strlen()` — стандартная функция Си для подсчета длины строки

<sup>181</sup>Пример базируется на исходнике отсюда: [http://www.strchr.com/sse2\\_optimised\\_strlen](http://www.strchr.com/sse2_optimised_strlen).

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        _BitScanForward(&pos, mask);
        len += (size_t)pos;

        break;
    }
    s += sizeof(__m128i);
    len += sizeof(__m128i);
};

return len;
}

```

Компилируем в MSVC 2010 с опцией /Ox:

Листинг 1.398: Оптимизирующий MSVC 2010

```

_pos$75552 = -4          ; size = 4
_str$ = 8              ; size = 4
?strlen_sse2@@YAIPBD@Z PROC ; strlen_sse2

    push    ebp
    mov     ebp, esp
    and     esp, -16          ; ffffffff0H
    mov     eax, DWORD PTR _str$[ebp]
    sub     esp, 12          ; 0000000cH
    push    esi
    mov     esi, eax
    and     esi, -16          ; ffffffff0H
    xor     edx, edx
    mov     ecx, eax
    cmp     esi, eax
    je      SHORT $LN4@strlen_sse
    lea     edx, DWORD PTR [eax+1]
    npad    3 ; выровнять следующую метку
$LL11@strlen_sse:
    mov     cl, BYTE PTR [eax]
    inc     eax
    test    cl, cl
    jne     SHORT $LL11@strlen_sse
    sub     eax, edx
    pop     esi
    mov     esp, ebp
    pop     ebp
    ret     0
$LN4@strlen_sse:
    movdqa  xmm1, XMMWORD PTR [eax]
    pxor    xmm0, xmm0
    pcmpeqb xmm1, xmm0
    pmovmskb eax, xmm1
    test    eax, eax
    jne     SHORT $LN9@strlen_sse
$LL3@strlen_sse:
    movdqa  xmm1, XMMWORD PTR [ecx+16]
    add     ecx, 16          ; 00000010H

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    pcmpeqb   xmm1, xmm0
    add      edx, 16                ; 00000010H
    pmovmskb eax, xmm1
    test     eax, eax
    je      SHORT $LL3@strlen_sse
$LN9@strlen_sse:
    bsf     eax, eax
    mov     ecx, eax
    mov     DWORD PTR _pos$75552[esp+16], eax
    lea    eax, DWORD PTR [ecx+edx]
    pop     esi
    mov     esp, ebp
    pop     ebp
    ret     0
?strlen_sse2@@YAIPBD@Z ENDP      ; strlen_sse2

```

Как это работает? Прежде всего, нужно определиться с целью этой ф-ции. Она вычисляет длину Си-строки, но можно сказать иначе — её задача это поиск нулевого байта, а затем вычисление его позиции относительно начала строки.

Итак, прежде всего, мы проверяем указатель `str`, выровнен ли он по 16-байтной границе. Если нет, то мы вызовем обычную реализацию `strlen()`.

Далее мы загружаем по 16 байт в регистр XMM1 при помощи команды `MOVDQA`.

Наблюдательный читатель может спросить, почему в этом месте мы не можем использовать `MOVDQU`, которая может загружать откуда угодно невзирая на факт, выровнен ли указатель?

Да, можно было бы сделать вот как: если указатель выровнен, загружаем используя `MOVDQA`, иначе используем работающую чуть медленнее `MOVDQU`.

Однако здесь кроется не сразу заметная проблема, которая проявляется вот в чем:

В **OS!** линии [Windows NT](#) (и не только), память выделяется страницами по 4 KiB (4096 байт). Каждый win32-процесс якобы имеет в наличии 4 GiB, но на самом деле, только некоторые части этого адресного пространства присоединены к реальной физической памяти. Если процесс обратится к блоку памяти, которого не существует, сработает исключение. Так работает **VM!**<sup>182</sup>.

Так вот, функция, читающая сразу по 16 байт, имеет возможность нечаянно вылезти за границу выделенного блока памяти. Предположим, **OS!** выделила программе 8192 (0x2000) байт по адресу 0x008c0000. Таким образом, блок занимает байты с адреса 0x008c0000 по 0x008c1fff включительно.

За этим блоком, то есть начиная с адреса 0x008c2000 нет вообще ничего, т.е. **OS!** не выделяла там память. Обращение к памяти начиная с этого адреса вызовет исключение.

И предположим, что программа хранит некую строку из, скажем, пяти символов почти в самом конце блока, что не является преступлением:

<sup>182</sup>[wikipedia](#)

0x008c1ff8	'h'
0x008c1ff9	'e'
0x008c1ffa	'l'
0x008c1ffb	'l'
0x008c1ffc	'o'
0x008c1ffd	'\x00'
0x008c1ffe	здесь случайный мусор
0x008c1fff	здесь случайный мусор

В обычных условиях, программа вызывает `strlen()` передав ей указатель на строку 'hello' лежащую по адресу 0x008c1ff8. `strlen()` будет читать по одному байту до 0x008c1ffd, где ноль, и здесь она закончит работу.

Теперь, если мы напишем свою реализацию `strlen()` читающую сразу по 16 байт, с любого адреса, будь он выровнен по 16-байтной границе или нет, `MOVVDQU` попытается загрузить 16 байт с адреса 0x008c1ff8 по 0x008c2008, и произойдет исключение. Это ситуация которой, конечно, хочется избежать.

Поэтому мы будем работать только с адресами, выровненными по 16 байт, что в сочетании со знанием что размер страницы **OS!** также, как правило, выровнен по 16 байт, даст некоторую гарантию что наша функция не будет пытаться читать из мест в невыделенной памяти.

Вернемся к нашей функции.

`_mm_setzero_si128()` — это макрос, генерирующий рхог `xmm0`, `xmm0` — инструкция просто обнуляет регистр `XMM0`.

`_mm_load_si128()` — это макрос для `MOVDDQA`, он просто загружает 16 байт по адресу из указателя в `XMM1`.

`_mm_cmpeq_epi8()` — это макрос для `PCMPQVB`, это инструкция, которая побайтово сравнивает значения из двух `XMM` регистров.

И если какой-то из байт равен другому, то в результирующем значении будет выставлено на месте этого байта `0xff`, либо `0`, если байты не были равны.

Например:

`XMM1: 0x11223344556677880000000000000000`

`XMM0: 0x11ab3444007877881111111111111111`

После исполнения `rcmpqvb xmm1, xmm0`, регистр `XMM1` содержит:

`XMM1: 0xff0000ff0000ffff0000000000000000`

Эта инструкция в нашем случае, сравнивает каждый 16-байтный блок с блоком состоящим из 16-и нулевых байт, выставленным в `XMM0` при помощи рхог `xmm0`, `xmm0`.

Следующий макрос `_mm_movemask_epi8()` — это инструкция `PMOVMASKB`.

Она очень удобна как раз для использования в паре с `PCMPQVB`.

`pmovmskb eax, xmm1`



Эта инструкция выставит самый первый бит EAX в единицу, если старший бит первого байта в регистре XMM1 является единицей. Иными словами, если первый байт в регистре XMM1 является 0xff, то первый бит в EAX будет также единицей, иначе нулем.

Если второй байт в регистре XMM1 является 0xff, то второй бит в EAX также будет единицей. Иными словами, инструкция отвечает на вопрос, «какие из байт в XMM1 имеют старший бит равный 1, или больше 0x7f?» В результате приготовит 16 бит и запишет в EAX. Остальные биты в EAX обнулятся.

Кстати, не забывайте также вот о какой особенности нашего алгоритма.

На вход может прийти 16 байт вроде:

15	14	13	12	11	10	9	3	2	1	0
'h'	'e'	'l'	'l'	'o'	0	мусор			0	мусор

Это строка 'hello', после нее терминирующий ноль, затем немного мусора в памяти.

Если мы загрузим эти 16 байт в XMM1 и сравним с нулевым XMM0, то в итоге получим такое <sup>183</sup>:

XMM1: 0x0000ff0000000000000000ff0000000000

Это означает, что инструкция сравнения обнаружила два нулевых байта, что и не удивительно.

PMOVMASKB в нашем случае подготовит EAX вот так:  
0b0010000000100000.

Совершенно очевидно, что далее наша функция должна учитывать только первый встретившийся нулевой бит и игнорировать все остальное.

Следующая инструкция — BSF (*Bit Scan Forward*). Это инструкция находит самый младший бит во втором операнде и записывает его позицию в первый операнд.

EAX=0b0010000000100000

После исполнения этой инструкции `bsf eax, eax`, `eax` в EAX будет 5, что означает, что единица найдена в пятой позиции (считая с нуля).

Для использования этой инструкции, в MSVC также имеется макрос `_BitScanForward`.

А дальше все просто. Если нулевой байт найден, его позиция прибавляется к тому что мы уже насчитали и возвращается результат.

Почти всё.

Кстати, следует также отметить, что компилятор MSVC сгенерировал два тела цикла сразу, для оптимизации.

Кстати, в SSE 4.2 (который появился в Intel Core i7) все эти манипуляции со строками могут быть еще проще: [http://www.strchr.com/strcmp\\_and\\_strlen\\_using\\_sse\\_4.2](http://www.strchr.com/strcmp_and_strlen_using_sse_4.2)

<sup>183</sup>Здесь используется порядок с **MSB!** до **LSB!**<sup>184</sup>.

## 1.37. 64 бита

### 1.37.1. x86-64

Это расширение x86-архитектуры до 64 бит.

С точки зрения начинающего reverse engineer-а, наиболее важные отличия от 32-битного x86 это:

- Почти все регистры (кроме FPU и SIMD) расширены до 64-бит и получили префикс R-. И еще 8 регистров добавлено. В итоге имеются эти **GPR!**-ы: RAX, RBX, RCX, RDX, RBP, RSP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15.

К ним также можно обращаться так же, как и прежде. Например, для доступа к младшим 32 битам RAX можно использовать EAX:

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
RAX <sup>x64</sup>							
				EAX			
						AX	
				AH		AL	

У новых регистров R8-R15 также имеются их *младшие части*: R8D-R15D (младшие 32-битные части), R8W-R15W (младшие 16-битные части), R8L-R15L (младшие 8-битные части).

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
R8							
				R8D			
						R8W	
				R8L			

Удвоено количество SIMD-регистров: с 8 до 16: XMM0-XMM15.

- В win64 передача всех параметров немного иная, это немного похоже на fastcall (?? (стр. ??)). Первые 4 аргумента записываются в регистры RCX, RDX, R8, R9, а остальные — в стек. Вызывающая функция также должна подготовить место из 32 байт чтобы вызываемая функция могла сохранить там первые 4 аргумента и использовать эти регистры по своему усмотрению. Короткие функции могут использовать аргументы прямо из регистров, но большие функции могут сохранять их значения на будущее.

Соглашение System V AMD64 ABI (Linux, \*BSD, Mac OS X)[Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)]<sup>185</sup> также напоминает fastcall, использует 6 регистров RDI, RSI, RDX, RCX, R8, R9 для первых шести аргументов. Остальные передаются через стек.

<sup>185</sup>Также доступно здесь: <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>

См. также в соответствующем разделе о способах передачи аргументов через стек (?? (стр. ??)).

- `int` в Си/Си++ остается 32-битным для совместимости.
- Все указатели теперь 64-битные.

Из-за того, что регистров общего пользования теперь вдвое больше, у компиляторов теперь больше свободного места для маневра, называемого [register allocation](#). Для нас это означает, что в итоговом коде будет меньше локальных переменных.

Для примера, функция вычисляющая первый S-блок алгоритма шифрования DES, она обрабатывает сразу 32/64/128/256 значений, в зависимости от типа `DES_type` (`uint32`, `uint64`, SSE2 или AVX), методом `bitslice DES` (больше об этом методе читайте здесь (?? (стр. ??))):

```

/*
 * Generated S-box files.
 *
 * This software may be modified, redistributed, and used for any purpose,
 * so long as its origin is acknowledged.
 *
 * Produced by Matthew Kwan - March 1998
 */

#ifdef _WIN64
#define DES_type unsigned __int64
#else
#define DES_type unsigned int
#endif

void
s1 (
    DES_type    a1,
    DES_type    a2,
    DES_type    a3,
    DES_type    a4,
    DES_type    a5,
    DES_type    a6,
    DES_type    *out1,
    DES_type    *out2,
    DES_type    *out3,
    DES_type    *out4
) {
    DES_type    x1, x2, x3, x4, x5, x6, x7, x8;
    DES_type    x9, x10, x11, x12, x13, x14, x15, x16;
    DES_type    x17, x18, x19, x20, x21, x22, x23, x24;
    DES_type    x25, x26, x27, x28, x29, x30, x31, x32;
    DES_type    x33, x34, x35, x36, x37, x38, x39, x40;
    DES_type    x41, x42, x43, x44, x45, x46, x47, x48;
    DES_type    x49, x50, x51, x52, x53, x54, x55, x56;

    x1 = a3 & ~a5;
    x2 = x1 ^ a4;

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
x3 = a3 & ~a4;
x4 = x3 | a5;
x5 = a6 & x4;
x6 = x2 ^ x5;
x7 = a4 & ~a5;
x8 = a3 ^ a4;
x9 = a6 & ~x8;
x10 = x7 ^ x9;
x11 = a2 | x10;
x12 = x6 ^ x11;
x13 = a5 ^ x5;
x14 = x13 & x8;
x15 = a5 & ~a4;
x16 = x3 ^ x14;
x17 = a6 | x16;
x18 = x15 ^ x17;
x19 = a2 | x18;
x20 = x14 ^ x19;
x21 = a1 & x20;
x22 = x12 ^ ~x21;
*out2 ^= x22;
x23 = x1 | x5;
x24 = x23 ^ x8;
x25 = x18 & ~x2;
x26 = a2 & ~x25;
x27 = x24 ^ x26;
x28 = x6 | x7;
x29 = x28 ^ x25;
x30 = x9 ^ x24;
x31 = x18 & ~x30;
x32 = a2 & x31;
x33 = x29 ^ x32;
x34 = a1 & x33;
x35 = x27 ^ x34;
*out4 ^= x35;
x36 = a3 & x28;
x37 = x18 & ~x36;
x38 = a2 | x3;
x39 = x37 ^ x38;
x40 = a3 | x31;
x41 = x24 & ~x37;
x42 = x41 | x3;
x43 = x42 & ~a2;
x44 = x40 ^ x43;
x45 = a1 & ~x44;
x46 = x39 ^ ~x45;
*out1 ^= x46;
x47 = x33 & ~x9;
x48 = x47 ^ x39;
x49 = x4 ^ x36;
x50 = x49 & ~x5;
x51 = x42 | x18;
x52 = x51 ^ a5;
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

x53 = a2 & ~x52;
x54 = x50 ^ x53;
x55 = a1 | x54;
x56 = x48 ^ ~x55;
*out3 ^= x56;
}

```

Здесь много локальных переменных. Конечно, далеко не все они будут в локальном стеке. Компилируем обычным MSVC 2008 с опцией /Ox:

Листинг 1.399: Оптимизирующий MSVC 2008

```

PUBLIC     _s1
; Function compile flags: /Ogtpy
_TEXT     SEGMENT
_x6$ = -20           ; size = 4
_x3$ = -16           ; size = 4
_x1$ = -12           ; size = 4
_x8$ = -8            ; size = 4
_x4$ = -4            ; size = 4
_a1$ = 8             ; size = 4
_a2$ = 12            ; size = 4
_a3$ = 16            ; size = 4
_x33$ = 20           ; size = 4
_x7$ = 20            ; size = 4
_a4$ = 20            ; size = 4
_a5$ = 24            ; size = 4
tv326 = 28           ; size = 4
_x36$ = 28           ; size = 4
_x28$ = 28           ; size = 4
_a6$ = 28            ; size = 4
_out1$ = 32          ; size = 4
_x24$ = 36           ; size = 4
_out2$ = 36          ; size = 4
_out3$ = 40          ; size = 4
_out4$ = 44          ; size = 4
_s1      PROC
    sub     esp, 20                ; 00000014H
    mov     edx, DWORD PTR _a5$[esp+16]
    push   ebx
    mov     ebx, DWORD PTR _a4$[esp+20]
    push   ebp
    push   esi
    mov     esi, DWORD PTR _a3$[esp+28]
    push   edi
    mov     edi, ebx
    not    edi
    mov     ebp, edi
    and    edi, DWORD PTR _a5$[esp+32]
    mov     ecx, edx
    not    ecx
    and    ebp, esi
    mov     eax, ecx
    and    eax, esi

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
and    ecx, ebx
mov    DWORD PTR _x1$[esp+36], eax
xor    eax, ebx
mov    esi, ebp
or     esi, edx
mov    DWORD PTR _x4$[esp+36], esi
and    esi, DWORD PTR _a6$[esp+32]
mov    DWORD PTR _x7$[esp+32], ecx
mov    edx, esi
xor    edx, eax
mov    DWORD PTR _x6$[esp+36], edx
mov    edx, DWORD PTR _a3$[esp+32]
xor    edx, ebx
mov    ebx, esi
xor    ebx, DWORD PTR _a5$[esp+32]
mov    DWORD PTR _x8$[esp+36], edx
and    ebx, edx
mov    ecx, edx
mov    edx, ebx
xor    edx, ebp
or     edx, DWORD PTR _a6$[esp+32]
not    ecx
and    ecx, DWORD PTR _a6$[esp+32]
xor    edx, edi
mov    edi, edx
or     edi, DWORD PTR _a2$[esp+32]
mov    DWORD PTR _x3$[esp+36], ebp
mov    ebp, DWORD PTR _a2$[esp+32]
xor    edi, ebx
and    edi, DWORD PTR _a1$[esp+32]
mov    ebx, ecx
xor    ebx, DWORD PTR _x7$[esp+32]
not    edi
or     ebx, ebp
xor    edi, ebx
mov    ebx, edi
mov    edi, DWORD PTR _out2$[esp+32]
xor    ebx, DWORD PTR [edi]
not    eax
xor    ebx, DWORD PTR _x6$[esp+36]
and    eax, edx
mov    DWORD PTR [edi], ebx
mov    ebx, DWORD PTR _x7$[esp+32]
or     ebx, DWORD PTR _x6$[esp+36]
mov    edi, esi
or     edi, DWORD PTR _x1$[esp+36]
mov    DWORD PTR _x28$[esp+32], ebx
xor    edi, DWORD PTR _x8$[esp+36]
mov    DWORD PTR _x24$[esp+32], edi
xor    edi, ecx
not    edi
and    edi, edx
mov    ebx, edi
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
and    ebx, ebp
xor    ebx, DWORD PTR _x28$[esp+32]
xor    ebx, eax
not    eax
mov    DWORD PTR _x33$[esp+32], ebx
and    ebx, DWORD PTR _a1$[esp+32]
and    eax, ebp
xor    eax, ebx
mov    ebx, DWORD PTR _out4$[esp+32]
xor    eax, DWORD PTR [ebx]
xor    eax, DWORD PTR _x24$[esp+32]
mov    DWORD PTR [ebx], eax
mov    eax, DWORD PTR _x28$[esp+32]
and    eax, DWORD PTR _a3$[esp+32]
mov    ebx, DWORD PTR _x3$[esp+36]
or     edi, DWORD PTR _a3$[esp+32]
mov    DWORD PTR _x36$[esp+32], eax
not    eax
and    eax, edx
or     ebx, ebp
xor    ebx, eax
not    eax
and    eax, DWORD PTR _x24$[esp+32]
not    ebp
or     eax, DWORD PTR _x3$[esp+36]
not    esi
and    ebp, eax
or     eax, edx
xor    eax, DWORD PTR _a5$[esp+32]
mov    edx, DWORD PTR _x36$[esp+32]
xor    edx, DWORD PTR _x4$[esp+36]
xor    ebp, edi
mov    edi, DWORD PTR _out1$[esp+32]
not    eax
and    eax, DWORD PTR _a2$[esp+32]
not    ebp
and    ebp, DWORD PTR _a1$[esp+32]
and    edx, esi
xor    eax, edx
or     eax, DWORD PTR _a1$[esp+32]
not    ebp
xor    ebp, DWORD PTR [edi]
not    ecx
and    ecx, DWORD PTR _x33$[esp+32]
xor    ebp, ebx
not    eax
mov    DWORD PTR [edi], ebp
xor    eax, ecx
mov    ecx, DWORD PTR _out3$[esp+32]
xor    eax, DWORD PTR [ecx]
pop    edi
pop    esi
xor    eax, ebx
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    pop     ebp
    mov     DWORD PTR [ecx], eax
    pop     ebx
    add     esp, 20
    ret     0
_s1      ENDP

```

5 переменных компилятору пришлось разместить в локальном стеке.

Теперь попробуем то же самое только в 64-битной версии MSVC 2008:

Листинг 1.400: Оптимизирующий MSVC 2008

```

a1$ = 56
a2$ = 64
a3$ = 72
a4$ = 80
x36$1$ = 88
a5$ = 88
a6$ = 96
out1$ = 104
out2$ = 112
out3$ = 120
out4$ = 128
s1      PROC
$LN3:
    mov     QWORD PTR [rsp+24], rbx
    mov     QWORD PTR [rsp+32], rbp
    mov     QWORD PTR [rsp+16], rdx
    mov     QWORD PTR [rsp+8], rcx
    push   rsi
    push   rdi
    push   r12
    push   r13
    push   r14
    push   r15
    mov     r15, QWORD PTR a5$[rsp]
    mov     rcx, QWORD PTR a6$[rsp]
    mov     rbp, r8
    mov     r10, r9
    mov     rax, r15
    mov     rdx, rbp
    not    rax
    xor    rdx, r9
    not    r10
    mov     r11, rax
    and    rax, r9
    mov     rsi, r10
    mov     QWORD PTR x36$1$[rsp], rax
    and    r11, r8
    and    rsi, r8
    and    r10, r15
    mov     r13, rdx
    mov     rbx, r11

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```
xor    rbx, r9
mov    r9, QWORD PTR a2$[rsp]
mov    r12, rsi
or     r12, r15
not    r13
and    r13, rcx
mov    r14, r12
and    r14, rcx
mov    rax, r14
mov    r8, r14
xor    r8, rbx
xor    rax, r15
not    rbx
and    rax, rdx
mov    rdi, rax
xor    rdi, rsi
or     rdi, rcx
xor    rdi, r10
and    rbx, rdi
mov    rcx, rdi
or     rcx, r9
xor    rcx, rax
mov    rax, r13
xor    rax, QWORD PTR x36$1$[rsp]
and    rcx, QWORD PTR a1$[rsp]
or     rax, r9
not    rcx
xor    rcx, rax
mov    rax, QWORD PTR out2$[rsp]
xor    rcx, QWORD PTR [rax]
xor    rcx, r8
mov    QWORD PTR [rax], rcx
mov    rax, QWORD PTR x36$1$[rsp]
mov    rcx, r14
or     rax, r8
or     rcx, r11
mov    r11, r9
xor    rcx, rdx
mov    QWORD PTR x36$1$[rsp], rax
mov    r8, rsi
mov    rdx, rcx
xor    rdx, r13
not    rdx
and    rdx, rdi
mov    r10, rdx
and    r10, r9
xor    r10, rax
xor    r10, rbx
not    rbx
and    rbx, r9
mov    rax, r10
and    rax, QWORD PTR a1$[rsp]
xor    rbx, rax
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
mov    rax, QWORD PTR out4$[rsp]
xor    rbx, QWORD PTR [rax]
xor    rbx, rcx
mov    QWORD PTR [rax], rbx
mov    rbx, QWORD PTR x36$1$[rsp]
and    rbx, rbp
mov    r9, rbx
not    r9
and    r9, rdi
or     r8, r11
mov    rax, QWORD PTR out1$[rsp]
xor    r8, r9
not    r9
and    r9, rcx
or     rdx, rbp
mov    rbp, QWORD PTR [rsp+80]
or     r9, rsi
xor    rbx, r12
mov    rcx, r11
not    rcx
not    r14
not    r13
and    rcx, r9
or     r9, rdi
and    rbx, r14
xor    r9, r15
xor    rcx, rdx
mov    rdx, QWORD PTR a1$[rsp]
not    r9
not    rcx
and    r13, r10
and    r9, r11
and    rcx, rdx
xor    r9, rbx
mov    rbx, QWORD PTR [rsp+72]
not    rcx
xor    rcx, QWORD PTR [rax]
or     r9, rdx
not    r9
xor    rcx, r8
mov    QWORD PTR [rax], rcx
mov    rax, QWORD PTR out3$[rsp]
xor    r9, r13
xor    r9, QWORD PTR [rax]
xor    r9, r8
mov    QWORD PTR [rax], r9
pop    r15
pop    r14
pop    r13
pop    r12
pop    rdi
pop    rsi
ret    0
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

s1	ENDP
----	------

Компилятор ничего не выделил в локальном стеке, а x36 это синоним для a5. Кстати, существуют процессоры с еще большим количеством **GPR!**, например, Itanium — 128 регистров.

### 1.37.2. ARM

64-битные инструкции появились в ARMv8.

### 1.37.3. Числа с плавающей запятой

О том как происходит работа с числами с плавающей запятой в x86-64, читайте здесь: ?? (стр. ??).

### 1.37.4. Критика 64-битной архитектуры

Некоторые люди иногда сетуют на то что указатели теперь 64-битные: ведь теперь для хранения всех указателей нужно в 2 раза больше места в памяти, в т.ч. и в кэш-памяти, не смотря на то что x64-процессоры могут адресовать только 48 бит внешней **RAM!**<sup>186</sup>.

Указатели уже настолько вышли из моды, что мне приходится вступать по этому поводу в споры. Если говорить о моем 64-разрядном компьютере, то, если действительно заботиться о производительности моего компьютера, мне приходится признать, что лучше отказаться от использования указателей, поскольку на моей машине 64-битные регистры, но всего 2 гигабайта оперативной памяти. Поэтому у указателя никогда не бывает больше 32 значащих битов. Но каждый раз, когда я использую указатель, это стоит мне 64 бита, и это удваивает размер моей структуры данных. Более того, это еще идет и в кэш-память, и половины кэш-памяти как не бывало, а за это приходится платить - кэш-память дорогая.

Поэтому я на самом деле пытаюсь сейчас пробовать новые варианты, то есть мне приходится вместо указателей использовать массивы. Я создаю сложные макросы, то есть создаю видимость использования указателей, хотя на самом деле их не использую.

( Дональд Кнут в “Кодеры за работой. Размышления о ремесле программиста”. )

Некоторые люди делают свои аллокаторы памяти. Интересен случай с CryptoMiniSat<sup>187</sup>. Эта программа довольно редко использует более 4GiB памяти, но она очень

<sup>186</sup> **RAM!**

<sup>187</sup> <https://github.com/msoos/cryptominisat/>

активно использует указатели. Так что, на 32-битной платформе она требовала меньше памяти, чем на 64-битной. Чтобы справиться с этой проблемой, автор создал свой аллокатор (в файлах *clauseallocator.(h|cpp)*), который позволяет иметь доступ к выделенной памяти используя 32-битные идентификаторы вместо 64-битных указателей.

## 1.38. Работа с числами с плавающей запятой (SIMD)

Разумеется, FPU остался в x86-совместимых процессорах в то время, когда ввели расширения **SIMD!**.

**SIMD!**-расширения (SSE2) позволяют удобнее работать с числами с плавающей запятой.

Формат чисел остается тот же (IEEE 754).

Так что современные компиляторы (включая те, что компилируют под x86-64) обычно используют **SIMD!**-инструкции вместо FPU-инструкций.

Это, можно сказать, хорошая новость, потому что работать с ними легче.

Примеры будем использовать из секции о FPU: ?? (стр. ??).

### 1.38.1. Простой пример

```
#include <stdio.h>

double f (double a, double b)
{
    return a/3.14 + b*4.1;
};

int main()
{
    printf ("%f\n", f(1.2, 3.4));
};
```

#### x64

Листинг 1.401: Оптимизирующий MSVC 2012 x64

```
__real@4010666666666666 DQ 0401066666666666r ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr ; 3.14

a$ = 8
b$ = 16
f      PROC
      divsd  xmm0, QWORD PTR __real@40091eb851eb851f
      mulsd  xmm1, QWORD PTR __real@4010666666666666
      addsd  xmm0, xmm1
      ret   0
f      ENDP
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Собственно, входные значения с плавающей запятой передаются через регистры XMM0–XMM3, а остальные — через стек <sup>188</sup>.

*a* передается через XMM0, *b* — через XMM1. Но XMM-регистры (как мы уже знаем из секции о **SIMD!** ?? (стр. ??)) 128-битные, а значения типа *double*— 64-битные, так что используется только младшая половина регистра.

DIVSD это SSE-инструкция, означает «Divide Scalar Double-Precision Floating-Point Values», и просто делит значение типа *double* на другое, лежащие в младших половинах операндов.

Константы закодированы компилятором в формате IEEE 754.

MULSD и ADDSD работают так же, только производят умножение и сложение.

Результат работы функции типа *double* функция оставляет в регистре XMM0.

Как работает неоптимизирующий MSVC:

Листинг 1.402: MSVC 2012 x64

```

__real@4010666666666666 DQ 0401066666666666r    ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr  ; 3.14

a$ = 8
b$ = 16
f    PROC
    movsdx  QWORD PTR [rsp+16], xmm1
    movsdx  QWORD PTR [rsp+8], xmm0
    movsdx  xmm0, QWORD PTR a$[rsp]
    divsd   xmm0, QWORD PTR __real@40091eb851eb851f
    movsdx  xmm1, QWORD PTR b$[rsp]
    mulsd   xmm1, QWORD PTR __real@4010666666666666
    addsd   xmm0, xmm1
    ret     0
f    ENDP

```

Чуть более избыточно. Входные аргументы сохраняются в «shadow space» (?? (стр. ??)), причем, только младшие половины регистров, т.е. только 64-битные значения типа *double*. Результат работы компилятора GCC точно такой же.

## x86

Скомпилируем этот пример также и под x86. MSVC 2012 даже генерируя под x86, использует SSE2-инструкции:

Листинг 1.403: Неоптимизирующий MSVC 2012 x86

```

tv70 = -8        ; size = 8
_a$ = 8          ; size = 8
_b$ = 16         ; size = 8
_f    PROC
    push    ebp

```

<sup>188</sup>[MSDN: Parameter Passing](#)

```

mov    ebp, esp
sub    esp, 8
movsd  xmm0, QWORD PTR _a$[ebp]
divsd  xmm0, QWORD PTR __real@40091eb851eb851f
movsd  xmm1, QWORD PTR _b$[ebp]
mulsd  xmm1, QWORD PTR __real@4010666666666666
addsd  xmm0, xmm1
movsd  QWORD PTR tv70[ebp], xmm0
fld    QWORD PTR tv70[ebp]
mov    esp, ebp
pop    ebp
ret    0
_f    ENDP

```

Листинг 1.404: Оптимизирующий MSVC 2012 x86

```

tv67 = 8      ; size = 8
_a$ = 8      ; size = 8
_b$ = 16     ; size = 8
_f    PROC
movsd  xmm1, QWORD PTR _a$[esp-4]
divsd  xmm1, QWORD PTR __real@40091eb851eb851f
movsd  xmm0, QWORD PTR _b$[esp-4]
mulsd  xmm0, QWORD PTR __real@4010666666666666
addsd  xmm1, xmm0
movsd  QWORD PTR tv67[esp-4], xmm1
fld    QWORD PTR tv67[esp-4]
ret    0
_f    ENDP

```

Код почти такой же, правда есть пара отличий связанных с соглашениями о вызовах:

- 1) аргументы передаются не в XMM-регистрах, а через стек, как и прежде, в примерах с FPU (?? (стр. ??));
- 2) результат работы функции возвращается через ST(0) — для этого он через стек (через локальную переменную tv) копируется из XMM-регистра в ST(0).

Попробуем оптимизированный пример в OllyDbg:

The screenshot displays the OllyDbg interface with the following components:

- Assembly Window:** Shows assembly code for the CPU - main thread, module simple. Key instructions include:
  - MOVSD XMM1, QWORD PTR SS:[ESP+4]
  - MOVSD XMM1, QWORD PTR DS:[13320C0]
  - MOVSD XMM0, QWORD PTR SS:[ESP+0C]
  - MOVSD XMM0, QWORD PTR DS:[13320D0]
  - ADDSD XMM1, XMM0
  - FLO QWORD PTR SS:[ESP+4], XMM1
  - FLO QWORD PTR SS:[ESP+4]
  - RETN
  - SUB ESP, 10
  - MOVSD XMM0, QWORD PTR DS:[13320C8]
  - MOVSD XMM0, QWORD PTR SS:[ESP+8], XMM0
  - MOVSD XMM0, QWORD PTR DS:[13320B8]
  - MOVSD XMM0, QWORD PTR SS:[ESP], XMM0
  - CALL 01331000
  - FSTP QWORD PTR SS:[ESP+8]
  - ADD ESP, 8
  - PUSH OFFSET 01333000
  - CALL QWORD PTR DS:[&MSUCR110.printf]
  - ADD ESP, 0C
  - XOR EAX, EAX
  - RETN
  - MOV EAX, 5A40
  - CMP WORD PTR DS:[<STRUCT IMAGE\_DOS\_HEAD
  - JE SHORT 01331082
  - XOR EAX, EAX
  - JMP SHORT 013310B6
  - MOV ECX, DWORD PTR DS:[133003C]
  - CMP DWORD PTR DS:[ECX+<STRUCT IMAGE\_DOS
  - JNE SHORT 0133107E
  - MOV EAX, 10B
  - CMP WORD PTR DS:[ECX+1330018] 0x
- Registers (FPU) Window:** Shows the state of floating-point registers. XMM0 and XMM1 are highlighted with values:
  - XMM0: 0.0
  - XMM1: 1.2000000000000000
- Hex Dump Window:** Shows memory addresses and their corresponding hex values. The address 013320C0 is highlighted, showing the value 3.1400000000000000.

Рис. 1.113: OllyDbg: MOVSD загрузила значение  $a$  в XMM1

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

The screenshot displays the OllyDbg interface with the following components:

- Assembly View:** Shows instructions starting at address 01331000. A key instruction is `F20F5E00 C02 DIVSD XMM1, QWORD PTR DS:[13320C0]`, which results in a float value of 3.4000. Other instructions include `F20F1042 0 MOVSD XMM0, QWORD PTR SS:[ESP+0C]`, `F20F5905 D02 HULSD XMM0, QWORD PTR DS:[13320D0]`, and `F20F59C3 ADDSD XMM1, XMM0`.
- Registers (FPU):** Shows the EAX register with the value 68F88634. Other registers like ECX, EDI, and ESP are also visible.
- Stack:** Shows the current stack frame with a return address of 0017FC08. The stack contains several zeroed-out values.

Рис. 1.114: OllyDbg: DIVSD вычислила частное и оставила его в XMM1

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



The screenshot displays the OllyDbg interface with three main panes:

- Assembly Window:** Shows assembly instructions for the CPU's main thread. Key instructions include:
  - `F20F104C24 0 MOVSD XMM1, QWORD PTR SS:[ESP+4]`
  - `F20F5E00 C02 DIVSD XMM1, QWORD PTR DS:[13320C0]`
  - `F20F104424 0 MOVSD XMM0, QWORD PTR SS:[ESP+0C1]`
  - `F20F5905 D02 MULSD XMM0, QWORD PTR DS:[13320D0]`
  - `F20F50C3 ADDSD XMM1, XMM0`
  - `F20F114C24 0 MOVSD QWORD PTR SS:[ESP+4], XMM1`
  - `004424 04 FLD QWORD PTR SS:[ESP+4]`
  - `F20F1005 C82 SUB ESP, 10`
  - `F20F114424 0 MOVSD QWORD PTR SS:[ESP+8], XMM0`
  - `F20F1005 B82 MOVSD XMM0, QWORD PTR DS:[13320B8]`
  - `F20F110424 0 MOVSD QWORD PTR SS:[ESP], XMM0`
  - `ES 0FFFFFFF CALL 01331000`
  - `D05C24 08 FSTP QWORD PTR SS:[ESP+8]`
  - `89C4 08 ADD ESP, 8`
  - `66 0030301 PUSH OFFSET 01333000`
  - `FE 5 3020330 CALL QWORD PTR DS:[&MSUCR110.printf<*>]`
  - `89C4 0C ADD ESP, 0C`
  - `39C0 XOR EAX, EAX`
  - `B8 405A0000 MOV EAX, 5A40`
  - `66:3905 0000 CMP WORD PTR DS:[<STRUCT IMAGE_DOS_HEAD`
  - `74 04 JLE SHORT 01331082`
  - `33C0 XOR EAX, EAX`
  - `EB 34 JMP SHORT 013310B6`
  - `8B00 3C00330 MOV ECX, DWORD PTR DS:[133003C]`
  - `81B9 0000330 CMP QWORD PTR DS:[ECX+<STRUCT IMAGE_DOS`
  - `75 0F JNE SHORT 0133107E`
  - `B8 0B010000 MOV EAX, 10B`
  - `66:3931 1300 CMP WORD PTR DS:[ECX+1330018], 0x`
- Registers (FPU) Window:** Shows the state of floating-point registers. XMM0 is highlighted with a value of `13.940000000000000`. Other registers like XMM1 through XMM7 are shown as `0.0`. The control word (CW) is `027F`.
- Hex Dump Window:** Shows the memory dump starting at address `01333000`. The first few bytes are `66 0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00`, which correspond to the `MOVSD` instruction shown in the assembly window.

Рис. 1.115: OllyDbg: MULSD вычислила произведение и оставила его в XMM0

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

The screenshot displays the OllyDbg interface with the following components:

- Assembly Window:** Shows assembly instructions with their addresses and hex values. Key instructions include:
  - 01331020: `F20F114C24 08 MOVSD QWORD PTR SS:[ESP+4],XMM1` (highlighted in red)
  - 01331021: `004424 04 FLD QWORD PTR SS:[ESP+4]`
  - 01331022: `C3 RETN`
  - 01331023: `93EC 10 SUB ESP,10`
  - 01331024: `F20F114424 08 MOVSD QWORD PTR DS:[13320C8]`
  - 01331025: `F20F1005 B82 MOVSD QWORD PTR DS:[ESP+8],XMM0`
  - 01331026: `F20F1005 B82 MOVSD QWORD PTR DS:[13320B8]`
  - 01331027: `F20F110424 MOVSD QWORD PTR SS:[ESP],XMM0`
  - 01331028: `ES 00FFFFFF CALL 01331000`
  - 01331029: `D05C24 08 FSTP QWORD PTR SS:[ESP+8]`
  - 01331030: `83C4 08 ADD ESP,8`
  - 01331031: `68 00303301 PUSH OFFSET 01333000`
  - 01331032: `FF15 00203301 CALL QWORD PTR DS:[&MSUCR110.printf]`
  - 01331033: `83C4 0C ADD ESP,0C`
  - 01331034: `33C0 XOR EAX,EAX`
  - 01331035: `C3 RETN`
  - 01331036: `CC INT3`
  - 01331037: `CC INT3`
  - 01331038: `CC INT3`
  - 01331039: `CC INT3`
  - 01331040: `CC INT3`
  - 01331041: `B8 405A0000 MOV EAX,5A40`
  - 01331042: `66:3905 0000 CMP WORD PTR DS:[<STRUCT IMAGE_DOS_HEAD`
  - 01331043: `74 04 JGE SHORT 01331082`
  - 01331044: `33C0 XOR EAX,EAX`
  - 01331045: `EB 34 JMP SHORT 013310B6`
  - 01331046: `8B00 3C003301 MOV ECX,QWORD PTR DS:[133003C]`
  - 01331047: `81B9 00003301 CMP QWORD PTR DS:[ECX+<STRUCT IMAGE_DOS`
  - 01331048: `75 04 JNE SHORT 0133107E`
  - 01331049: `B8 00010000 MOV EAX,100`
  - 01331050: `66:3901 13001 CMP WORD PTR DS:[ECX+13300181] 0x`
- Registers (FPU) Window:** Shows the state of various registers. The floating-point registers (FPU) are visible, with values like `0.0` and `13.940000000000000` for XMM0 and XMM1.
- Stack Window:** Shows memory addresses and hex dumps. The stack pointer is at `0017FBC4`. The stack contains several `00` bytes, indicating a return from a function.

Рис. 1.116: OllyDbg: ADDSD прибавила значение в XMM0 к XMM1

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

The screenshot displays the OllyDbg interface with three main panes:

- Assembly Window:** Shows assembly instructions for the function `simple.0133102A`. Key instructions include `MOVSD XMM0, QWORD PTR SS:[ESP+4]`, `DIVSD XMM1, QWORD PTR DS:[13320C0]`, `MULSD XMM0, QWORD PTR DS:[13320D0]`, `ADDSD XMM1, XMM0`, and `FLD QWORD PTR SS:[ESP+4], XMM1`. The `FLD` instruction is highlighted, and its result is visible in the registers window.
- Registers (FPU) Window:** Shows the state of the floating-point unit registers. `ST(0)` contains the value `valid 14.322165605095539930`. Other registers like `ST(1)` through `ST(7)` are empty. The `FST` register shows a condition code of `Cond 0 0 0 0` and an error code of `Err 0 0 0 0 0 0 0 0 (GT)`.
- Stack Window:** Shows the stack frame for `simple.0133102A`. The return address is `0017FBC0`. The stack contains several `00` bytes, indicating a return value of `0`.

Рис. 1.117: OllyDbg: FLD оставляет результат функции в ST(0)

Видно, что OllyDbg показывает XMM-регистры как пары чисел в формате *double*, но используется только младшая часть.

Должно быть, OllyDbg показывает их именно так, потому что сейчас исполняются SSE2-инструкции с суффиксом -SD.

Но конечно же, можно переключить отображение значений в регистрах и посмотреть содержимое как 4 *float*-числа или просто как 16 байт.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

## 1.38.2. Передача чисел с плавающей запятой в аргументах

```
#include <math.h>
#include <stdio.h>

int main ()
{
    printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

    return 0;
}
```

Они передаются в младших половинах регистров XMM0-XMM3.

Листинг 1.405: Оптимизирующий MSVC 2012 x64

```
$SG1354 DB      '32.01 ^ 1.54 = %lf', 0aH, 00H

__real@40400147ae147ae1 DQ 040400147ae147ae1r    ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r  ; 1.54

main    PROC
        sub     rsp, 40                                ; 00000028H
        movsdx  xmm1, QWORD PTR __real@3ff8a3d70a3d70a4
        movsdx  xmm0, QWORD PTR __real@40400147ae147ae1
        call    pow
        lea    rcx, OFFSET FLAT:$SG1354
        movaps  xmm1, xmm0
        movd    rdx, xmm1
        call    printf
        xor     eax, eax
        add     rsp, 40                                ; 00000028H
        ret     0
main    ENDP
```

Инструкции MOVSDX нет в документации от Intel и AMD (?? (стр. ??)), там она называется просто MOVSD. Таким образом, в процессорах x86 две инструкции с одинаковым именем (о второй: ?? (стр. ??)). Возможно, в Microsoft решили избежать путаницы и переименовали инструкцию в MOVSDX. Она просто загружает значение в младшую половину XMM-регистра.

Функция pow() берет аргументы из XMM0 и XMM1, и возвращает результат в XMM0. Далее он перекладывается в RDX для printf(). Почему? Может быть, это потому что printf()— функция с переменным количеством аргументов?

Листинг 1.406: Оптимизирующий GCC 4.4.6 x64

```
.LC2:
    .string "32.01 ^ 1.54 = %lf\n"

main:
    sub     rsp, 8
    movsd  xmm1, QWORD PTR .LC0[rip]
    movsd  xmm0, QWORD PTR .LC1[rip]
    call   pow
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; результат сейчас в XMM0
mov     edi, OFFSET FLAT:.LC2
mov     eax, 1 ; количество переданных векторных регистров
call   printf
xor     eax, eax
add     rsp, 8
ret
.LC0:
        .long   171798692
        .long   1073259479
.LC1:
        .long   2920577761
        .long   1077936455

```

GCC работает понятнее. Значение для `printf()` передается в XMM0. Кстати, вот тот случай, когда в EAX для `printf()` записывается 1 — это значит, что будет передан один аргумент в векторных регистрах, так того требует стандарт [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)]<sup>189</sup>.

### 1.38.3. Пример со сравнением

```

#include <stdio.h>

double d_max (double a, double b)
{
    if (a>b)
        return a;

    return b;
};

int main()
{
    printf ("%f\n", d_max (1.2, 3.4));
    printf ("%f\n", d_max (5.6, -4));
};

```

## x64

Листинг 1.407: Оптимизирующий MSVC 2012 x64

```

a$ = 8
b$ = 16
d_max PROC
        comisd   xmm0, xmm1
        ja      SHORT $LN2@d_max
        movaps  xmm0, xmm1
$LN2@d_max:

```

<sup>189</sup>Также доступно здесь: <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

fatret 0
d_max  ENDP

```

Оптимизирующий MSVC генерирует очень понятный код.

Инструкция COMISD это «Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS». Собственно, это она и делает.

Неоптимизирующий MSVC генерирует более избыточно, но тоже всё понятно:

Листинг 1.408: MSVC 2012 x64

```

a$ = 8
b$ = 16
d_max  PROC
        movsdx  QWORD PTR [rsp+16], xmm1
        movsdx  QWORD PTR [rsp+8], xmm0
        movsdx  xmm0, QWORD PTR a$[rsp]
        comisd  xmm0, QWORD PTR b$[rsp]
        jbe     SHORT $LN1@d_max
        movsdx  xmm0, QWORD PTR a$[rsp]
        jmp     SHORT $LN2@d_max
$LN1@d_max:
        movsdx  xmm0, QWORD PTR b$[rsp]
$LN2@d_max:
        fatret  0
d_max  ENDP

```

А вот GCC 4.4.6 дошел в оптимизации дальше и применил инструкцию MAXSD («Return Maximum Scalar Double-Precision Floating-Point Value»), которая просто выбирает максимальное значение!

Листинг 1.409: Оптимизирующий GCC 4.4.6 x64

```

d_max:
        maxsd   xmm0, xmm1
        ret

```

**x86**

Скомпилируем этот пример в MSVC 2012 с включенной оптимизацией:

Листинг 1.410: Оптимизирующий MSVC 2012 x86

```
_a$ = 8      ; size = 8
_b$ = 16     ; size = 8
_d_max PROC
    movsd    xmm0, QWORD PTR _a$(esp-4)
    comisd   xmm0, QWORD PTR _b$(esp-4)
    jbe      SHORT $LN1@d_max
    fld      QWORD PTR _a$(esp-4)
    ret     0
$LN1@d_max:
    fld      QWORD PTR _b$(esp-4)
    ret     0
_d_max ENDP
```

Всё то же самое, только значения *a* и *b* берутся из стека, а результат функции оставляется в ST(0).

Если загрузить этот пример в OllyDbg, увидим, как инструкция COMISD сравнивает значения и устанавливает/сбрасывает флаги CF и PF:

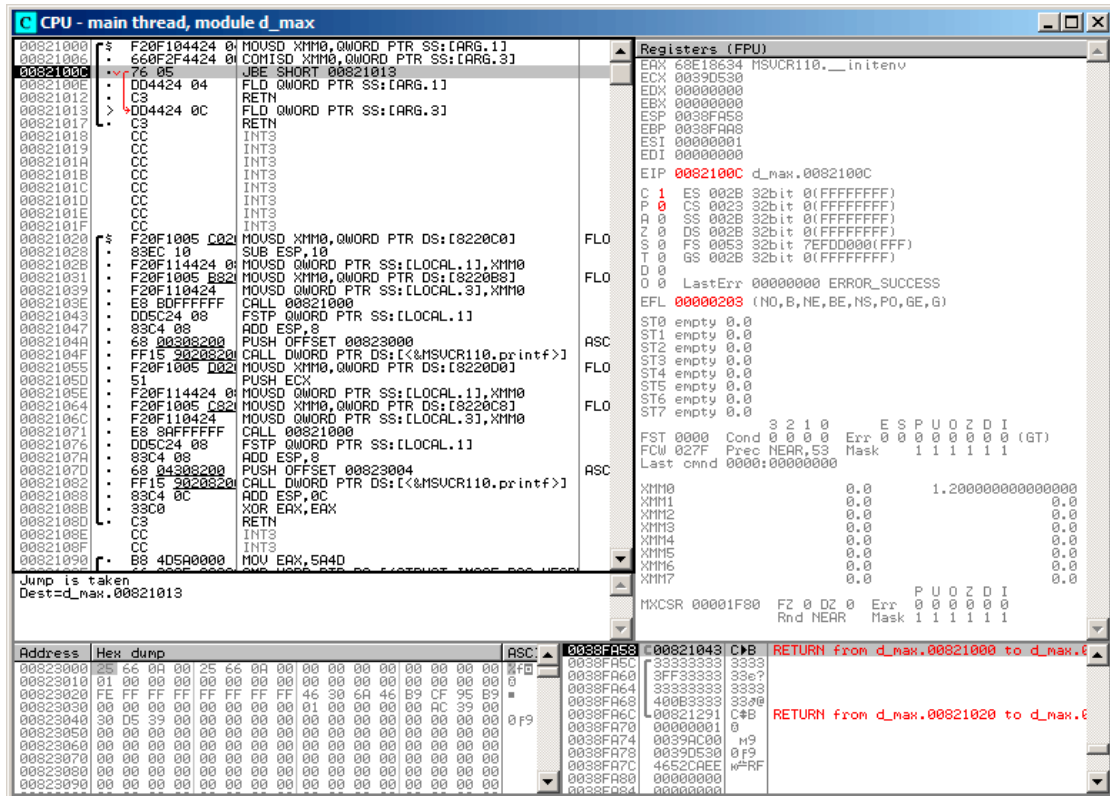


Рис. 1.118: OllyDbg: COMISD изменила флаги CF и PF

### 1.38.4. Вычисление машинного эпсилона: x64 и SIMD

Вернемся к примеру «вычисление машинного эпсилона» для *double* листинг.??.

Теперь скомпилируем его для x64:

Листинг 1.411: Оптимизирующий MSVC 2012 x64

```

v$ = 8
calculate_machine_epsilon PROC
    movsdx  QWORD PTR v$[rsp], xmm0
    movaps  xmm1, xmm0
    inc     QWORD PTR v$[rsp]
    movsdx  xmm0, QWORD PTR v$[rsp]
    subsd  xmm0, xmm1
    ret     0
calculate_machine_epsilon ENDP

```

Нет способа прибавить 1 к значению в 128-битном XMM-регистре, так что его нужно в начале поместить в память.

Впрочем, есть инструкция *ADDSD (Add Scalar Double-Precision Floating-Point Values)*,

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



которая может прибавить значение к младшей 64-битной части XMM-регистра игнорируя старшую половину, но наверное MSVC 2012 пока недостаточно хорош для этого

[190](#).

Так или иначе, значение затем перезагружается в XMM-регистр и происходит вычитание.

SUBSD это «Subtract Scalar Double-Precision Floating-Point Values», т.е. операция производится над младшей 64-битной частью 128-битного XMM-регистра. Результат возвращается в регистре XMM0.

### 1.38.5. И снова пример генератора случайных чисел

Вернемся к примеру «пример генератора случайных чисел» листинг.??.

Если скомпилировать это в MSVC 2012, компилятор будет использовать SIMD-инструкции для FPU.

Листинг 1.412: Оптимизирующий MSVC 2012

```

__real@3f800000 DD 03f800000r    ; 1

tv128 = -4
_tmp$ = -4
?float_rand@@YAMXZ PROC
    push    ecx
    call    ?my_rand@@YAIXZ
; EAX=псевдослучайное значение
    and     eax, 8388607        ; 007ffffFH
    or      eax, 1065353216    ; 3f800000H
; EAX=псевдослучайное значение & 0x007ffff | 0x3f800000
; сохранить его в локальном стеке:
    mov     DWORD PTR _tmp$[esp+4], eax
; перезагрузить его как число с плавающей точкой:
    movss  xmm0, DWORD PTR _tmp$[esp+4]
; вычесть 1.0:
    subss  xmm0, DWORD PTR __real@3f800000
; переместить значение в ST0 поместив его во временную переменную...
    movss  DWORD PTR tv128[esp+4], xmm0
; ... и затем перезагрузив её в ST0:
    fld    DWORD PTR tv128[esp+4]
    pop    ecx
    ret    0
?float_rand@@YAMXZ ENDP

```

У всех инструкций суффикс -SS, это означает «Scalar Single».

«Scalar» означает, что только одно значение хранится в регистре.

«Single»<sup>191</sup> означает, что это тип *float*.

<sup>190</sup>В качестве упражнения, вы можете попробовать переработать этот код, чтобы избавиться от использования локального стека.

<sup>191</sup>Т.е., single precision.

### 1.38.6. Итог

Во всех приведенных примерах, в XMM-регистрах используется только младшая половина регистра, там хранится значение в формате IEEE 754.

Собственно, все инструкции с суффиксом -SD («Scalar Double-Precision») — это инструкции для работы с числами с плавающей запятой в формате IEEE 754, хранящиеся в младшей 64-битной половине XMM-регистра.

Всё удобнее чем это было в FPU, видимо, сказывается тот факт, что расширения SIMD развивались не так стихийно, как FPU в прошлом.

Стековая модель регистров не используется.

Если вы попытаетесь заменить в этих примерах *double* на *float*, то инструкции будут использоваться те же, только с суффиксом -SS («Scalar Single-Precision»), например, MOVSS, COMISS, ADDSS, и т. д.

«Scalar» означает, что SIMD-регистр будет хранить только одно значение, вместо нескольких.

Инструкции, работающие с несколькими значениями в регистре одновременно, имеют «Packed» в названии.

Нужно также обратить внимание, что SSE2-инструкции работают с 64-битными числами (*double*) в формате IEEE 754, в то время как внутреннее представление в FPU — 80-битные числа.

Поэтому ошибок округления (*round-off error*) в FPU может быть меньше чем в SSE2, как следствие, можно сказать, работа с FPU может давать более точные результаты вычислений.

## 1.39. Кое-что специфичное для ARM

### 1.39.1. Знак номера (#) перед числом

Компилятор Keil, **IDA!** и objdump предваряет все числа знаком номера («#»), например:

листинг.???. Но когда GCC 4.9 выдает результат на языке ассемблера, он так не делает, например:

листинг.???

Так что листинги для ARM в этой книге в каком-то смысле перемешаны.

Трудно сказать, как правильнее. Должно быть, всякий должен придерживаться тех правил, которые приняты в той среде, в которой он работает.

### 1.39.2. Режимы адресации

В ARM64 возможна такая инструкция:

```
ldr    x0, [x29,24]
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

И это означает прибавить 24 к значению в X29 и загрузить значение по этому адресу. Обратите внимание что 24 внутри скобок.

А если снаружи скобок, то весь смысл меняется:

```
ldr    w4, [x1],28
```

Это означает, загрузить значение по адресу в X1, затем прибавить 28 к X1.

ARM позволяет прибавлять некоторую константу к адресу, с которого происходит загрузка, либо вычитать.

Причем, позволяет это делать до загрузки или после.

Такого режима адресации в x86 нет, но он есть в некоторых других процессорах, даже на PDP-11.

Существует байка, что режимы пре-инкремента, пост-инкремента, пре-декремента и пост-декремента адреса в PDP-11, были «виновны» в появлении таких конструкций языка Си (который разрабатывался на PDP-11) как `*ptr++`, `*++ptr`, `*ptr--`, `*--ptr`. Кстати, это является трудно запоминаемой особенностью в Си.

Дела обстоят так:

термин в Си	термин в ARM	выражение Си	как это работает
Пост-инкремент	post-indexed addressing	<code>*ptr++</code>	использовать значение <code>*ptr</code> , затем инкремент указателя <code>ptr</code>
Пост-декремент	post-indexed addressing	<code>*ptr--</code>	использовать значение <code>*ptr</code> , затем <b>декремент</b> указателя <code>ptr</code>
Пре-инкремент	pre-indexed addressing	<code>*++ptr</code>	инкремент указателя <code>ptr</code> , затем использовать значение <code>*ptr</code>
Пре-декремент	pre-indexed addressing	<code>*--ptr</code>	<b>декремент</b> указателя <code>ptr</code> , затем использовать значение <code>*ptr</code>

Pre-indexing маркируется как восклицательный знак в ассемблере ARM. Для примера, смотрите строку 2 в листинг.??

Деннис Ритчи (один из создателей ЯП Си) указывал, что, это, вероятно, придумал Кен Томпсон (еще один создатель Си), потому что подобная возможность процессора имелась еще в PDP-7 <sup>192</sup>, [Dennis M. Ritchie, *The development of the C language*, (1993)]<sup>193</sup>. Таким образом, компиляторы с ЯП Си на тот процессор, где это есть, могут использовать это.

Всё это очень удобно для работы с массивами.

<sup>192</sup>[http://yurichev.com/mirrors/C/c\\_dmr\\_postincrement.txt](http://yurichev.com/mirrors/C/c_dmr_postincrement.txt)

<sup>193</sup>Также доступно здесь: [pdf](#)

### 1.39.3. Загрузка констант в регистр

#### 32-битный ARM

Как мы уже знаем, все инструкции имеют длину в 4 байта в режиме ARM и 2 байта в режиме Thumb.

Как в таком случае записать в регистр 32-битное число, если его невозможно закодировать внутри одной инструкции?

Попробуем:

```
unsigned int f()
{
    return 0x12345678;
};
```

Листинг 1.413: GCC 4.6.3 -O3 Режим ARM

```
f:
    ldr    r0, .L2
    bx    lr
.L2:
    .word 305419896 ; 0x12345678
```

Т.е., значение 0x12345678 просто записано в памяти отдельно и загружается, если нужно.

Но можно обойтись и без дополнительного обращения к памяти.

Листинг 1.414: GCC 4.6.3 -O3 -march=armv7-a (Режим ARM)

```
movw    r0, #22136      ; 0x5678
movt    r0, #4660      ; 0x1234
bx      lr
```

Видно, что число загружается в регистр по частям, в начале младшая часть (при помощи инструкции MOVW), затем старшая (при помощи MOVT).

Следовательно, нужно 2 инструкции в режиме ARM, чтобы записать 32-битное число в регистр.

Это не так уж и страшно, потому что в реальном коде не так уж и много констант (кроме 0 и 1).

Значит ли это, что это выполняется медленнее чем одна инструкция, как две инструкции?

Вряд ли, наверняка современные процессоры ARM наверняка умеют распознавать такие последовательности и исполнять их быстро.

А **IDA!** легко распознает подобные паттерны в коде и дизассемблирует эту функцию как:

```
MOV     R0, 0x12345678
BX      LR
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

**ARM64**

```
uint64_t f()
{
    return 0x12345678ABCDEF01;
};
```

Листинг 1.415: GCC 4.9.1 -O3

```
mov    x0, 61185    ; 0xef01
movk   x0, 0xabcd, lsl 16
movk   x0, 0x5678, lsl 32
movk   x0, 0x1234, lsl 48
ret
```

MOVK означает «MOV Keep», т.е. она записывает 16-битное значение в регистр, не трогая при этом остальные биты. Суффикс LSL сдвигает значение в каждом случае влево на 16, 32 и 48 бит. Сдвиг происходит перед загрузкой. Таким образом, нужно 4 инструкции, чтобы записать в регистр 64-битное значение.

**Записать числа с плавающей точкой в регистр**

Некоторые числа можно записывать в D-регистр при помощи только одной инструкции.

Например:

```
double a()
{
    return 1.5;
};
```

Листинг 1.416: GCC 4.9.1 -O3 + objdump

```
0000000000000000 <a>:
 0: 1e6f1000    fmov    d0, #1.5000000000000000e+000
 4: d65f03c0    ret
```

Число 1.5 действительно было закодировано в 32-битной инструкции.

Но как? В ARM64, инструкцию FM0V есть 8 бит для кодирования некоторых чисел с плавающей запятой.

В [ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, (2013)]<sup>194</sup> алгоритм называется VFPEExpandImm().

Это также называется *minifloat*<sup>195</sup>. Мы можем попробовать разные: 30.0 и 31.0 компилятору удастся закодировать, а 32.0 уже нет, для него придется выделять 8 байт в памяти и записать его там в формате IEEE 754:

<sup>194</sup>Также доступно здесь: [http://yurichev.com/mirrors/ARMv8-A\\_Architecture\\_Reference\\_Manual\\_\(Issue\\_A.a\).pdf](http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf)

<sup>195</sup>wikipedia

```
double a()
{
    return 32;
};
```

Листинг 1.417: GCC 4.9.1 -O3

```
a:
    ldr    d0, .LC0
    ret
.LC0:
    .word  0
    .word  1077936128
```

### 1.39.4. Релоки в ARM64

Как известно, в ARM64 инструкции 4-байтные, так что записать длинное число в регистр одной инструкцией нельзя.

Тем не менее, файл может быть загружен по произвольному адресу в памяти, для этого релоки и нужны.

Больше о них (в связи с Win32 PE): ?? (стр. ??).

В ARM64 принят следующий метод: адрес формируется при помощи пары инструкций: ADRP и ADD.

Первая загружает в регистр адрес 4KiB-страницы, а вторая прибавляет остаток.

Скомпилируем пример из «Hello, world!» (листинг.??) в GCC (Linaro) 4.9 под win32:

Листинг 1.418: GCC (Linaro) 4.9 и objdump объектного файла

```
...>aarch64-linux-gnu-gcc.exe hw.c -c
...>aarch64-linux-gnu-objdump.exe -d hw.o
...
0000000000000000 <main>:
 0: a9bf7bfd      stp    x29, x30, [sp,#-16]!
 4: 910003fd      mov    x29, sp
 8: 90000000      adrp   x0, 0 <main>
 c: 91000000      add    x0, x0, #0x0
10: 94000000      bl     0 <printf>
14: 52800000      mov    w0, #0x0 // #0
18: a8c17bfd      ldp    x29, x30, [sp],#16
1c: d65f03c0      ret

...>aarch64-linux-gnu-objdump.exe -r hw.o
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

...
RELOCATION RECORDS FOR [.text]:
OFFSET          TYPE          VALUE
0000000000000008 R_AARCH64_ADR_PREL_PG_HI21 .rodata
000000000000000c R_AARCH64_ADD_ABS_L012_NC  .rodata
0000000000000010 R_AARCH64_CALL26  printf

```

Итак, в этом объектом файле три релока.

- Самый первый берет адрес страницы, отсекает младшие 12 бит и записывает оставшиеся старшие 21 в битовые поля инструкции ADRP. Это потому что младшие 12 бит кодировать не нужно, и в ADRP выделено место только для 21 бит.
- Второй — 12 бит адреса, относительного от начала страницы, в поля инструкции ADD.
- Последний, 26-битный, накладывается на инструкцию по адресу 0x10, где переход на функцию printf().

Все адреса инструкций в ARM64 (да и в ARM в режиме ARM) имеют нули в двух младших битах (потому что все инструкции имеют размер в 4 байта), так что нужно кодировать только старшие 26 бит из 28-битного адресного пространства ( $\pm 128\text{MB}$ ).

В слинкованном исполняемом файле релоков в этих местах нет: потому что там уже точно известно, где будет находиться строка «Hello!», и в какой странице, а также известен адрес функции puts().

И поэтому там, в инструкциях ADRP, ADD и BL, уже проставлены нужные значения (их проставил линкер во время компоновки):

Листинг 1.419: objdump исполняемого файла

```

0000000000400590 <main>:
400590:    a9bf7bfd      stp     x29, x30, [sp,#-16]!
400594:    910003fd      mov     x29, sp
400598:    90000000      adrp   x0, 400000 <_init-0x3b8>
40059c:    91192000      add     x0, x0, #0x648
4005a0:    97ffffa0      bl     400420 <puts@plt>
4005a4:    52800000      mov     w0, #0x0 // #0
4005a8:    a8c17bfd      ldp    x29, x30, [sp],#16
4005ac:    d65f03c0      ret

...

Contents of section .rodata:
400640 01000200 00000000 48656c6c 6f210000  ....Hello!..

```

В качестве примера, попробуем дизассемблировать инструкцию BL вручную. 0x97ffffa0 это 0b100101111111111111111111111111110100000. В соответствии с [ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, (2013)C5.6.26], imm26 это последние 26 бит:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

$imm26 = 0b111111111111111110100000$ . Это  $0x3FFFA0$ , но **MSB!** это 1, так что число отрицательное, мы можем вручную его конвертировать в удобный для нас вид. По правилам изменения знака, просто инвертируем все биты:  $(0b1011111=0x5F)$  и прибавляем 1 ( $0x5F+1=0x60$ ). Так что число в знаковом виде:  $-0x60$ . Умножим  $-0x60$  на 4 (потому что адрес записанный в опкоде разделен на 4): это  $-0x180$ . Теперь вычисляем адрес назначения:  $0x4005a0 + (-0x180) = 0x400420$  (пожалуйста заметьте: мы берем адрес инструкции BL, а не текущее значение **PC!**, которое может быть другим!). Так что адрес в итоге  $0x400420$ .

Больше о релоках связанных с ARM64: [ELF for the ARM 64-bit Architecture (AArch64), (2013)]<sup>196</sup>.

## 1.40. Кое-что специфичное для MIPS

### 1.40.1. Загрузка 32-битной константы в регистр

```
unsigned int f()
{
    return 0x12345678;
};
```

В MIPS, так же как и в ARM, все инструкции имеют размер 32 бита, так что невозможно закодировать 32-битную константу в инструкцию.

Так что приходится делать это используя по крайней мере две инструкции: первая загружает старшую часть 32-битного числа и вторая применяет операцию «ИЛИ», эффект от которой в том, что она просто выставляет младшие 16 бит целевого регистра:

Листинг 1.420: GCC 4.4.5 -O3 (вывод на ассемблере)

```
li    $2,305397760 # 0x12340000
j     $31
ori   $2,$2,0x5678 ; branch delay slot
```

**IDA!** знает о таких часто встречающихся последовательностях, так что для удобства, она показывает последнюю инструкцию ORI как псевдоинструкцию LI, которая якобы загружает полное 32-битное значение в регистр \$V0.

Листинг 1.421: GCC 4.4.5 -O3 (IDA)

```
lui   $v0, 0x1234
jr    $ra
li    $v0, 0x12345678 ; branch delay slot
```

В выводе на ассемблере от GCC есть псевдоинструкция LI, но на самом деле, там LUI («Load Upper Immediate»), загружающая 16-битное значение в старшую часть регистра.

Посмотрим в выводе *objdump*:

<sup>196</sup>Также доступно здесь: [http://infocenter.arm.com/help/topic/com.arm.doc.ihl0056b/IHI0056B\\_aaelf64.pdf](http://infocenter.arm.com/help/topic/com.arm.doc.ihl0056b/IHI0056B_aaelf64.pdf)



Листинг 1.422: objdump

```
00000000 <f>:
 0: 3c021234      lui    v0,0x1234
 4: 03e00008      jr     ra
 8: 34425678      ori    v0,v0,0x5678
```

### Загрузка 32-битной глобальной переменной в регистр

```
unsigned int global_var=0x12345678;

unsigned int f2()
{
    return global_var;
};
```

Тут немного иначе: LUI загружает старшие 16 бит из *global\_var* в \$2 (или \$V0) и затем LW загружает младшие 16 бит суммируя их с содержимым \$2:

Листинг 1.423: GCC 4.4.5 -O3 (вывод на ассемблере)

```
f2:
    lui    $2,%hi(global_var)
    lw     $2,%lo(global_var)($2)
    j     $31
    nop    ; branch delay slot

    ...

global_var:
    .word 305419896
```

**IDA!** знает о часто применяемой паре инструкций LUI/LW, так что она объединяет их в одну инструкцию LW:

Листинг 1.424: GCC 4.4.5 -O3 (IDA)

```
_f2:
    lw     $v0, global_var
    jr     $ra
    or     $at, $zero      ; branch delay slot

    ...

    .data
    .globl global_var
global_var: .word 0x12345678      # DATA XREF: _f2
```

Вывод *objdump* почти такой же, как ассемблерный вывод GCC. Посмотрим также релоки в объектном файле:

Листинг 1.425: objdump

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

objdump -D filename.o

...

0000000c <f2>:
   c: 3c020000      lui    v0,0x0
  10: 8c420000      lw     v0,0(v0)
  14: 03e00008      jr     ra
  18: 00200825      move  at,at ; branch delay slot
 1c: 00200825      move  at,at

Disassembly of section .data:

00000000 <global_var>:
   0: 12345678      beq   s1,s4,159e4 <f2+0x159d8>

...

objdump -r filename.o

...

RELOCATION RECORDS FOR [.text]:
OFFSET  TYPE          VALUE
0000000c R_MIPS_HI16   global_var
00000010 R_MIPS_L016   global_var

...

```

Можем увидеть, что адрес *global\_var* будет записываться прямо в инструкции LUI и LW во время загрузки исполняемого файла: старшая 16-битная часть *global\_var* записывается в первую инструкцию (LUI), младшая 16-битная часть — во вторую (LW).

## 1.40.2. Книги и прочие материалы о MIPS

Dominic Sweetman, *See MIPS Run, Second Edition*, (2010).

## Глава 2

# Важные фундаментальные вещи

В [Mathematical Recipes](https://math.recipes)<sup>1</sup> есть некоторые важные замечания о булевой алгебре и представлении знаковых чисел.

### 2.1. Целочисленные типы данных

Целочисленный тип данных (*integral*) это тип для значения, которое может быть сконvertировано в число. Это числа, перечисления (*enumerations*), булевы типы.

#### 2.1.1. Бит

Очевидное использования бит это булевы значения: 0 для *ложно/false* и 1 для *true/истинно*.

Набор булевых значений можно упаковать в **слово**: в 32-битном слове будет 32 булевых значения, и т. д. Этот метод также называется *bitmap* или *bitfield*.

Но есть очевидные накладные расходы: тасовка бит, изоляция оных, и т. д. В то время как использование **слова** (или типа *int*) для булевого значения это не экономично, но очень эффективно.

В среде Си/Си++, 0 это *false/ложно* и любое ненулевое значение это *true/истинно*. Например:

```
if (1234)
    printf ("это всегда будет исполняться\n");
else
    printf ("а это никогда\n");
```

<sup>1</sup><https://math.recipes>

Это популярный способ перечислить все символы в Си-строке:

```
char *input=...;

while(*input) // исполнять тело, если в *input character не ноль
{
    // делать что-то с *input
    input++;
};
```

### 2.1.2. Ниббл АКА nibble АКА nybble

**АКА!** полубайт, тетрада. Равняется 4-м битам.

Все эти термины в ходу и сегодня.

#### Двоично-десятичный код (BCD!<sup>2</sup>)

4-битные нибблы использовались в 4-битных процессорах, например, в легендарном Intel 4004 (который использовался в калькуляторах).

Интересно знать, что числа там представлялись в виде *binary-coded decimal* (**BCD!**). Десятичный 0 кодировался как 0b0000, десятичная 9 как 0b1001, а остальные значения не использовались. Десятичное 1234 представлялось как 0x1234. Конечно, этот способ не очень экономичный.

Тем не менее, он имеет одно преимущество: очень легко конвертировать значения из десятичного в **BCD!**-запакованное и назад. BCD-числа можно складывать, вычитать, и т. д., но нужна дополнительная корректировка. В x86 CPUs для этого есть редкие инструкции: AAA/DAA (adjust after addition: корректировка после сложения), AAS/DAS (adjust after subtraction: корректировка после вычитания), AAM (after multiplication: после умножения), AAD (after division: после деления).

Необходимость поддерживать **BCD!**-числа в CPU это причина, почему существуют флаги *half-carry flag* (флаг полупереноса) (в 8080/Z80) и *auxiliary flag* (вспомогательный флаг) (AF в x86): это флаг переноса, генерируемый после обработки младших 4-х бит. Флаг затем используется корректирующими инструкциями.

Тот факт, что числа легко конвертировать, привел к популярности этой книги: [Peter Abel, *IBM PC assembly language and programming* (1987)]. Но кроме этой книги, автор этих заметок, никогда не видел **BCD!**-числа на практике, исключая *magic numbers* (?? (стр. ??)), как, например, дата чье-то дня рождения, закодированная как 0x19791011 — это действительно запакованное **BCD!**-число.

На удивление, автор нашел использование чисел закодированных в **BCD!** в ПО SAP: <https://yurichev.com/blog/SAP/>. Некоторые числа, включая цены, кодируются в виде **BCD!** в базе. Вероятно, они использовали это для совместимости с каким-то древним ПО или железом?

<sup>2</sup>BCD!

Инструкции для **BCD!** в x86 часто использовались для других целей, использовались их недокументированные особенности, например:

```
cmp al,10
sbb al,69h
das
```

Этот запутанный код конвертирует число в пределах 0..15 в **ASCII!**-символ '0'..'9', 'A'..'F'.

## Z80

Z80 был клоном 8-битного Intel 8080 CPU, и из-за экономии места, он имеет 4-битный **ALU!**, т.е., каждая операция над двумя 8-битными числами происходит за два шага. Один из побочных эффектов в том, что легко генерировать *half-carry flag* (флаг полупереноса).

### 2.1.3. Байт

Байт, в первую очередь, применяется для хранения символов. 8-битные байты не всегда были популярны, как сейчас. Перфоленты для телетайпов имели 5 и 6 возможных дырок, это 5 или 6 бит на байт.

Чтобы подчеркнуть тот факт, что в байте 8 бит, байт иногда называется *октетом* (*octet*): по крайней мере *fetchmail* использует эту терминологию.

9-битные байты существовали на 36-битных архитектурах: 4 9-битных байта помещались в одно **слово**. Вероятно из-за этого, стандарты Си/Си++ говорят что в *char* должно быть *как минимум* 8 бит, но может быть и больше.

Например, в ранней документации к языку Си<sup>3</sup>, можно найти такое:

```
char one byte character (PDP-11, IBM360: 8 bits; H6070: 9 bits)
```

Под H6070, вероятно, подразумевается Honeywell 6070, с 36-битными словами.

### Стандартная ASCII-таблица

7-битная ASCII-таблица стандартная, которая содержит только 128 возможных символов. Раннее ПО для передачи e-мейлов работало только с 7-битными ASCII-символами, так что понадобился стандарт **MIME!**<sup>4</sup> для кодирования сообщений в нелатинских системах письменности. 7-битные ASCII коды дополнялись битом чётности, давая в итоге 8 бит.

*Data Encryption Standard* (**DES!**<sup>5</sup>) имеет 56-битный ключ, это 8 7-битных байт, оставляя место для бита чётности для каждого символа.

<sup>3</sup><https://yurichev.com/mirrors/C/bwk-tutor.html>

<sup>4</sup>**MIME!**

<sup>5</sup>**DES!**

Заучивать на память всю таблицу **ASCII** незачем, но можно запомнить интервалы. [0..0x1F] это управляющие символы (непечатные). [0x20..0x7E] это печатные. Коды начиная с 0x80 обычно используются для нелатинских систем письменности и/или псевдографики.

Некоторые важные коды, которые легко запомнить: 0 (конец Си-строки, '\0' в C/C++); 0xA или 10 (*line feed* (перевод строки), '\n' в C/C++); 0xD или 13 (*carriage return* (возврат каретки), '\r' в C/C++).

0x20 (пробел) также часто запоминается.

## 8-битные процессоры

x86 имеют возможность работать с байтами на уровне регистров (потому что они наследники 8-битного процессора 8080), а RISC как ARM и MIPS — нет.

### 2.1.4. Wide char

Это попытка поддерживать многоязычную среду расширяя байт до 16-и бит. Самый известный пример это ядро Windows NT и win32-функции с суффиксом *W*. Вот почему если закодировать обычный текст на английском, то каждый латинский символ в текстовой строке будет перемежаться с нулевым байтом. Эта кодировка также называется UCS-2 или UTF-16

Обычно, *wchar\_t* это синоним 16-битного типа данных *short*.

### 2.1.5. Знаковые целочисленные и беззнаковые

Некоторые люди могут удивляться, почему беззнаковые типы данных вообще существуют, т.к., любое беззнаковое число можно представить как знаковое. Да, но отсутствие бита знака в значении расширяет интервал в два раза. Следовательно, знаковый байт имеет интервал -128..127, а беззнаковый: 0..255. Еще одно преимущество беззнаковых типов данных это самодокументация: вы определяете переменную, которая не может принимать отрицательные значения.

Беззнаковые типы данных отсутствуют в Java, за что её критикуют. Трудно реализовать криптографические алгоритмы используя булевы операции над знаковыми типами.

Значения вроде 0xFFFFFFFF (-1) часто используются, в основном, как коды ошибок.

### 2.1.6. Слово (word)

Слово **слово** это неоднозначный термин, и обычно означает тип данных, помещающийся в **GPR**!. Байты практичны для символов, но непрактичны для арифметических расчетов.

Так что, многие процессоры имеют **GPR!** шириной 16, 32 или 64 бит. Даже 8-битные **CPU!** как 8080 и Z80 предлагают работать с парами 8-битными регистров, каждая пара формирует 16-битный псевдорегистр (*BC, DE, HL*, и т. д.). Z80 имеет некоторые возможности для работы с парами регистров, и это, в каком-то смысле, эмуляция 16-битного CPU.

В общем, если в рекламе CPU говорят о нем как о “*n*-битном процессоре”, это обычно означает, что он имеет *n*-битные **GPR!**.

Было время, когда в рекламе жестких дисков и модулей **RAM!** писали, что они имеют *n* килослов вместо *b* килобайт/мегабайт.

Например, *Apollo Guidance Computer* имел 2048 слов **RAM!**. Это был 16-битный компьютер, так что там было 4096 байт **RAM!**.

*TX-0* имел 64K 18-битных слов памяти на магнитных сердечниках, т.е., 64 килослов.

*DECSYSTEM-2060* мог иметь вплоть до 4096 килослов *твердотельной памяти* (т.е., жесткие диски, ленты, и т. д.). Это был 36-битный компьютер, так что это 18432 килобайта или 18 мегабайт.

В сущности, зачем нужны байты, если есть слова? Если только для работы с текстовыми строками. Почти во всех остальных случаях можно использовать слова.

---

*int* в Си/Си++ почти всегда связан со **словом**. (Кроме архитектуры AMD64, где *int* остался 32-битным, вероятно, ради лучшей обратной совместимости.)

*int* 16-битный на PDP-11 и старых компьютерах с MS-DOS. *int* 32-битный на VAX, и на x86 начиная с 80386, и т. д.

И даже более того, если в программе на Си/Си++ определение типа для переменной отсутствует, то по умолчанию подразумевается *int*. Вероятно, это наследие языка программирования В<sup>6</sup>.

---

**GPR!** это обычно самый быстрый контейнер для переменной, быстрее чем упакованный бит, и иногда даже быстрее упакованного байта (потому что нет нужды изолировать единственный бит/байт из **GPR!**). Даже если вы используете его как контейнер для счетчика в цикле, в интервале 0..99.

---

В языке ассемблера, **word** всё еще 16-битный для x86, потому что так было во времена 16-битного 8086. *Double word* 32-битный, *quad word* 64-битный. Вот почему 16-битные слова определяются при помощи DW в ассемблере на x86, для 32-битных используется DD и для 64-битных — DQ.

**Word** 32-битный для ARM, MIPS, и т. д., 16-битные типы данных называются здесь *half-word* (полуслово). Следовательно, *double word* на 32-битном RISC это 64-битный тип данных.

---

<sup>6</sup><http://yurichev.com/blog/typeless/>

В *GDB* такая терминология: *halfword* для 16-битных, *word* для 32-битных и *giant word* для 64-битных.

В 16-битной среде Си/Си++ на PDP-11 и MS-DOS был тип *long* шириной в 32 бита, вероятно, они имели ввиду *long word* или *long int*?

В 32-битных средах Си/Си++ имеется тип *long long* для типов данных шириной 64 бита.

Теперь вы видите, почему термин *слово* такой неоднозначный.

### Нужно ли использовать *int*?

Некоторые люди говорят о том, что тип *int* лучше не использовать вообще, потому что его неоднозначность приводит к ошибкам. Например, хорошо известная библиотека *lzbuf* использует тип *int* в одном месте, и всё работает нормально на 16-битной архитектуре. Но если она портируется на архитектуру с 32-битным *int*, она может падать: <http://yurichev.com/blog/lzbuf/>.

Более однозначные типы определены в файле *stdint.h*: *uint8\_t*, *uint16\_t*, *uint32\_t*, *uint64\_t*, и т. д.

Некоторые люди, как Дональд Э. Кнут, предлагают<sup>7</sup> более звучные слова для этих типов:

*byte/wyde/tetrabyte/tetra/octabyte/octa*. Но эти имена менее популярны чем ясные термины с включением символа *u* (*unsigned*) и числом прямо в названии типа.

### Компьютеры ориентированные на слово

Не смотря на неоднозначность термина *слово*, современные компьютеры всё ещё ориентированы на слово: **RAM!** и все уровни кэш-памяти организованы по словам а не байтам. Впрочем, в рекламе пишут о размере именно в байтах.

Доступ по адресу в памяти и кэш-памяти выровненный по границе слова зачастую быстрее, чем невыровненный.

При разработке структур данных, от которых ждут скорости и эффективности, всегда нужно учитывать длину *слова* CPU, на котором это будет исполняться. Иногда компилятор делает это за программиста, иногда нет.

#### 2.1.7. Регистр адреса

Для тех, кто был воспитан на 32-битных и/или 64-битных x86, и/или RISC 90-х годов, как ARM, MIPS, PowerPC, считается обычным, что шина адреса имеет такую же ширину как **GPR!** или *слово*. Тем не менее, на других архитектурах, ширина шины адреса может быть другой.

8-битный Z80 может адресовать  $2^{16}$  байт, используя пары 8-битных регистров, или специальные регистры (*IX*, *IY*). Регистры *SP* и *PC* также 16-битные.

<sup>7</sup><http://www-cs-faculty.stanford.edu/~uno/news98.html>



Суперкомпьютер Cray-1 имел 64-битные GPR, но 24-битные регистры для адресов, так что он мог адресовать  $2^{24}$  (16 мегаслов или 128 мегабайт). Память в 1970-ые была очень дорогой, и типичный Cray-1 имел 1048576 (0x100000) слов ОЗУ или 8МВ. Тогда зачем выделять целый 64-битный регистр для адреса или указателя?

Процессоры 8086/8088 имели крайне странную схему адресации: значения двух 16-битных регистров суммировались в очень странной манере, производя 20-битный адрес. Вероятно, то было что-то вроде игрушечной виртуализации (?? (стр. ??))? 8086 мог исполнять несколько программ (хотя и не одновременно).

Ранний ARM1 имеет интересный артефакт:

Another interesting thing about the register file is the PC register is missing a few bits. Since the ARM1 uses 26-bit addresses, the top 6 bits are not used. Because all instructions are aligned on a 32-bit boundary, the bottom two address bits in the PC are always zero. These 8 bits are not only unused, they are omitted from the chip entirely.

( <http://www.righto.com/2015/12/reverse-engineering-arm1-ancestor-of.html> )

Так что, значение где в двух младших битах что-то есть, невозможно записать в регистр PC просто физически. Также невозможно установить любой бит в старших 6 битах PC.

Архитектура x86-64 имеет 64-битные виртуальные указателя/адреса, но внутри адресная шина 48-битная (этого достаточно для адресации 256ТВ памяти).

## 2.1.8. Числа

Для чего используются числа?

Когда вы видите как некое число/числа меняются в регистре процесса, вы можете заинтересоваться, что это число значит. Это довольно важное качество реверс-инжинира, определять возможный тип данных по набору изменяемых чисел.

### Булевы значения

Если число меняется от 0 до 1 и назад, скорее всего, это значение имеет булевый тип данных.

### Счетчик циклов, индекс массива

Переменная увеличивающаяся с 0, как: 0, 1, 2, 3...— большая вероятность что это счетчик цикла и/или индекс массива.

## Знаковые числа

Если вы видите переменную, которая содержит очень маленькие числа, и иногда очень большие, как 0, 1, 2, 3, и 0xFFFFFFFF, 0xFFFFFFFFE, 0xFFFFFFFFD, есть шанс что это знаковая переменная в виде *дополнительного кода*, и последние три числа это -1, -2, -3.

## 32-битные числа

Существуют настолько большие числа, что для них даже существует специальная нотация (Knuth's up-arrow notation). Эти числа настолько большие, что им нет практического применения в инженерии, науке и математике.

Почти всем инженерам и ученым зачастую достаточно чисел в формате IEEE 754 в двойной точности, где максимальное значение близко к  $1.8 \cdot 10^{308}$ . (Для сравнения, количество атомов в наблюдаемой Вселенной оценивается от  $4 \cdot 10^{79}$  до  $4 \cdot 10^{81}$ .)

А в практическом программировании, верхний предел значительно ниже. Так было в эпоху MS-DOS: 16-битные *int* использовались почти везде (индексы массивов, счетчики циклов), в то время как 32-битные *long* использовались редко.

Во время появления x86-64, было решено оставить тип *int* 32-битным, вероятно, потому что необходимость использования 64-битного *int* еще меньше.

Я бы сказал, что 16-битные числа в интервале 0..65535, вероятно, наиболее используемые числа в программировании вообще.

Учитывая всё это, если вы видите необычно большое 32-битное значение вроде 0x87654321, большая вероятность, что это может быть:

- это всё еще может быть 16-битное число, но знаковое, между 0xFFFF8000 (-32768) и 0xFFFFFFFF (-1).
- адрес ячейки памяти (можно проверить используя в карте памяти в отладчике);
- запакованные байты (можно проверить визуально);
- битовые флаги;
- что-то связанное с (любительской) криптографией;
- *магическое число* (?? (стр. ??));
- число с плавающей точкой в формате IEEE 754 (тоже легко проверить).

Та же история и для 64-битных значений.

## ...так что, 16-битного *int* достаточно почти для всего?

Интересно заметить: в [Michael Abrash, *Graphics Programming Black Book*, 1997 глава 13] мы можем найти множество случаев, когда 16-битных переменных просто достаточно. В то же время, Майкл Абраш жалеет о том что в процессорах 80386 и 80486 маловато доступных регистров, так что он предлагает

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

хранить два 16-битных значения в одном 32-битном регистре и затем прокручивать его используя инструкцию ROR reg, 16 (на 80386 и позже) (ROL reg, 16 тоже будет работать) или BSWAP (на 80486 и позже).

Это нам напоминает как в Z80 был набор альтернативных регистров (с апострофом в конце), на которые CPU мог переключаться (и затем переключаться назад) используя инструкцию EXX.

## Размер буфера

Когда программисту нужно обозначить размер некоторого буфера, обычно используются значения вида  $2^x$  (512 байт, 1024, и т. д.). Значения вида  $2^x$  легко опознать (?? (стр. ??)) в десятичной, шестнадцатеричной и двоичной системе.

Но надо сказать что программисты также и люди со своей десятичной культурой. И иногда, в среде **DBMS!**<sup>8</sup>, размер текстовых полей в БД часто выбирается в виде числа  $10^x$ , как 100, 200. Они думают что-то вроде «Окей, 100 достаточно, погодите, лучше пусть будет 200». И они правы, конечно.

Максимальный размер типа данных VARCHAR2 в Oracle RDBMS это 4000 символов, а не 4096.

В этом нет ничего плохого, это просто еще одно место, где можно встретить числа вида  $10^x$ .

## Адрес

Всегда хорошая идея это держать в памяти примерную карту памяти процесса, который вы отлаживаете. Например, многие исполняемые файлы в win32 начинаются с 0x00401000, так что адрес вроде 0x00451230 скорее всего находится в секции с исполняемым кодом. Вы увидите адреса вроде этих в регистре EIP.

Стек обычно расположен где-то ниже.

Многие отладчики могут показывать карту памяти отлаживаемого процесса, например: ?? (стр. ??).

Если значение увеличивается с шагом 4 на 32-битной архитектуре или с шагом 8 на 64-битной, это вероятно сдвигающийся адрес некоторых элементов массива.

Важно знать что win32 не использует адреса ниже 0x10000, так что если вы видите какое-то число ниже этой константы, это не может быть адресом (см. также: <https://msdn.microsoft.com/en-us/library/ms810627.aspx>).

Так или иначе, многие отладчики могут показывать, является ли значение в регистре адресом чего-либо. OllyDbg также может показывать ASCII-строку, если значение является её адресом.

---

<sup>8</sup>**DBMS!**

## Битовые поля

Если вы видите как в значении один (или больше) бит меняются от времени к времени, как 0xABCD1234 → 0xABCD1434 и назад, это вероятно битовое поле (или *bitmap*).

## Запакованные байты

Когда *strcpy()* или *memcpy()* копирует буфер, они загружают/записывают 4 (или 8) байт одновременно, так что если строка содержит «4321» и будет скопирована в другое место, в какой-то момент вы увидите значение 0x31323334 в каком-либо регистре. Это 4 запакованных байта в одном 32-битном значении.

### 2.1.9. AND/OR/XOR как MOV

Инструкция `OR reg, 0xFFFFFFFF` выставляет все биты в 1, следовательно, не важно что было в регистре перед этим, его значение будет выставлено в -1. Инструкция `OR reg, -1` короче, чем `MOV reg, -1`, так что MSVC использует `OR` вместо последней, например: ?? (стр. ??).

Точно также, `AND reg, 0` всегда сбрасывает все биты, следовательно, работает как `MOV reg, 0`.

`XOR reg, reg`, не важно что было в регистре перед этим, сбрасывает все биты, и также работает как `MOV reg, 0`.

## 2.2. Endianness (порядок байт)

Endianness (порядок байт) это способ представления чисел в памяти.

### 2.2.1. Big-endian (от старшего к младшему)

Число 0x12345678 представляется в памяти так:

адрес в памяти	значение байта
+0	0x12
+1	0x34
+2	0x56
+3	0x78

CPU с таким порядком включают в себя Motorola 68k, IBM POWER.

### 2.2.2. Little-endian (от младшего к старшему)

Число 0x12345678 представляется в памяти так:

адрес в памяти	значение байта
+0	0x78
+1	0x56
+2	0x34
+3	0x12

CPU с таким порядком байт включают в себя Intel x86. Один важный пример использования little-endian в этой книге: ?? (стр. ??).

### 2.2.3. Пример

Возьмем big-endian Linux для MIPS заинсталлированный в QEMU <sup>9</sup>.

И скомпилируем этот простой пример:

```
#include <stdio.h>

int main()
{
    int v;

    v=123;

    printf ("%02X %02X %02X %02X\n",
           *(char*)&v,
           (((char*)&v)+1),
           (((char*)&v)+2),
           (((char*)&v)+3));
};
```

И запустим его:

```
root@debian-mips:~# ./a.out
00 00 00 7B
```

Это оно и есть. 0x7B это 123 в десятичном виде. В little-endian-архитектуре, 7B это первый байт (вы можете это проверить в x86 или x86-64), но здесь он последний, потому что старший байт идет первым.

Вот почему имеются разные дистрибутивы Linux для MIPS («mips» (big-endian) и «mipsel» (little-endian)). Программа скомпилированная для одного соглашения об endiannes, не сможет работать в OS использующей другое соглашение.

Еще один пример связанный с big-endian в MIPS в этой книге: ?? (стр. ??).

### 2.2.4. Bi-endian (переключаемый порядок)

CPU поддерживающие оба порядка, и его можно переключать, включают в себя ARM, PowerPC, SPARC, MIPS, **IA64!**<sup>10</sup>, и т. д.

<sup>9</sup>Доступен для скачивания здесь: <https://people.debian.org/~aurel32/qemu/mips/>

<sup>10</sup>**IA64!**

## 2.2.5. Конвертирование

Инструкция BSWAP может использоваться для конвертирования.

Сетевые пакеты TCP/IP используют соглашение big-endian, вот почему программа, работающая на little-endian архитектуре должна конвертировать значения.

Обычно, используются функции `htonl()` и `htons()`.

Порядок байт big-endian в среде TCP/IP также называется, «network byte order», а порядок байт на компьютере «host byte order». На архитектуре Intel x86, и других little-endian архитектурах, «host byte order» это little-endian, а вот на IBM POWER это может быть big-endian, так что на последней, `htonl()` и `htons()` не меняют порядок байт.

## 2.3. Память

Есть три основных типа памяти:

- Глобальная память **АКА!** «static memory allocation». Нет нужды явно выделять, выделение происходит просто при объявлении переменных/массивов глобально. Это глобальные переменные расположенные в сегменте данных или констант. Доступны глобально (поэтому считаются **анти-паттерном**). Не удобны для буферов/массивов, потому что должны иметь фиксированный размер. Переполнения буфера, случающиеся здесь, обычно перезаписывают переменные или буферы расположенные рядом в памяти. Пример в этой книге: ?? (стр. ??).
- Стек **АКА!** «allocate on stack», «выделить память в/на стеке». Выделение происходит просто при объявлении переменных/массивов локально в функции. Обычно это локальные для функции переменные. Иногда эти локальные переменные также доступны и для нисходящих функций (**callee**-функциям, если функция-**caller** передает указатель на переменную в функцию-**callee**). Выделение и освобождение очень быстрое, достаточно просто сдвига **SP!**.

Но также не удобно для буферов/массивов, потому что размер буфера фиксирован, если только не используется `alloca()` (?? (стр. ??)) (или массив с переменной длиной).

Переполнение буфера обычно перезаписывает важные структуры стека: ?? (стр. ??).

- Куча (*heap*) **АКА!** «dynamic memory allocation», «выделить память в куче». Выделение происходит при помощи вызова `malloc()/free()` или `new/delete` в Си++.

Самый удобный метод: размер блока может быть задан во время исполнения. Изменение размера возможно (при помощи `realloc()`), но может быть медленным.

Это самый медленный метод выделения памяти: аллокатор памяти должен поддерживать и обновлять все управляющие структуры во время выделения и освобождения. Переполнение буфера обычно перезаписывает

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

---

все эти структуры. Выделения в куче также ведут к проблеме утечек памяти: каждый выделенный блок должен быть явно освобожден, но кто-то может забыть об этом, или делать это неправильно. Еще одна проблема — это «использовать после освобождения» — использовать блок памяти после того как `free()` был вызван на нем, это тоже очень опасно. Пример в этой книге: ?? (стр. ??).

## 2.4. CPU

### 2.4.1. Предсказатели переходов

Некоторые современные компиляторы пытаются избавиться от инструкций условных переходов. Примеры в этой книге: ?? (стр. ??), ?? (стр. ??), ?? (стр. ??).

Это потому что предсказатель переходов далеко не всегда работает идеально, поэтому, компиляторы и стараются реже использовать переходы, если возможно.

Одна из возможностей — это условные инструкции в ARM (как `ADRcc`), а еще инструкция `CMOVcc` в x86.

### 2.4.2. Зависимости между данными

Современные процессоры способны исполнять инструкции одновременно (**OOE!**<sup>11</sup>), но для этого, внутри такой группы, результат одних не должен влиять на работу других. Следовательно, компилятор старается использовать инструкции с наименьшим влиянием на состояние процессора.

Вот почему инструкция `LEA` в x86 такая популярная — потому что она не модифицирует флаги процессора, а прочие арифметические инструкции — модифицируют.

## 2.5. Хеш-функции

Простейший пример это CRC32, алгоритм «более мощный» чем простая контрольная сумма, для проверки целостности данных. Невозможно восстановить оригинальный текст из хеша, там просто меньше информации: ведь текст может быть очень длинным, но результат CRC32 всегда ограничен 32 битами. Но CRC32 не надежна в криптографическом смысле: известны методы как изменить текст таким образом, чтобы получить нужный результат. Криптографические хеш-функции защищены от этого.

Такие функции как MD5, SHA1, и т. д., широко используются для хеширования паролей для хранения их в базе. Действительно: БД форума в интернете может и не хранить пароли (иначе злоумышленник получивший доступ к БД сможет узнать все пароли), а только хеши. К тому же, скрипту интернет-форума вовсе не обязательно знать ваш пароль, он только должен сверить его

---

<sup>11</sup>OOE!

хеш с тем что лежит в БД, и дать вам доступ если сверка проходит. Один из самых простых способов взлома — это просто перебирать все пароли и ждать пока результат будет такой же как тот что нам нужен. Другие методы намного сложнее.

### 2.5.1. Как работает односторонняя функция?

Односторонняя функция, это функция, которая способна превратить из одного значения другое, при этом невозможно (или трудно) проделать обратную операцию. Некоторые люди имеют трудности с пониманием, как это возможно. Рассмотрим очень простой пример.

У нас есть ряд из 10-и чисел в пределах 0..9, каждое встречается один раз, например:

4 6 0 1 3 5 7 8 9 2
---------------------

Алгоритм простейшей односторонней функции выглядит так:

- возьми число на нулевой позиции (у нас это 4);
- возьми число на первой позиции (у нас это 6);
- обменяй местами числа на позициях 4 и 6.

Отметим числа на позициях 4 и 6:

4 6 0 1 3 5 7 8 9 2
^  ^

Меняем их местами и получаем результат:

4 6 0 1 7 5 3 8 9 2
---------------------

Глядя на результат, и даже зная алгоритм функции, мы не можем однозначно восстановить изначальное положение чисел. Ведь первые два числа могли быть 0 и/или 1, и тогда именно они могли бы участвовать в обмене.

Это крайне упрощенный пример для демонстрации, настоящие односторонние функции могут быть значительно сложнее.



## Глава 3

# Более сложные примеры

### 3.1. Двойное отрицание

Популярный способ<sup>1</sup> сконvertировать ненулевое значение в 1 (или булево *true*) и 0 в 0 (или булево *false*) это *!!statement*:

```
int convert_to_bool(int a)
{
    return !!a;
};
```

Оптимизирующий GCC 5.4 x86:

```
convert_to_bool:
    mov     edx, DWORD PTR [esp+4]
    xor     eax, eax
    test   edx, edx
    setne  al
    ret
```

XOR всегда очищает возвращаемое значение в EAX, даже если SETNE не работает. Т.е., XOR устанавливает возвращаемое значение (по умолчанию) в 0.

Если входное значение не равно нулю (суффикс -NE в инструкции SET), тогда 1 заносится в AL, иначе AL не модифицируется.

Почему SETNE работает с младшей 8-битной частью регистра EAX? Потому что значение имеет только последний бит (0 или 1), а остальные биты были уже сброшены при помощи XOR.

Следовательно, этот код на Си/Си++ может быть переписан так:

```
int convert_to_bool(int a)
{
    if (a!=0)
```

<sup>1</sup>Хотя и спорный, потому что приводит к трудночитаемому коду

```

        return 1;
    else
        return 0;
};

```

...или даже:

```

int convert_to_bool(int a)
{
    if (a)
        return 1;
    else
        return 0;
};

```

Компиляторы, компилирующие для **CPU!** у которых нет инструкции близкой к SET, в этом случае, генерируют инструкции условного перехода, и т. д.

## 3.2. Использование const (const correctness)

Это незаслуженно малоиспользуемая возможность многих ЯП. Тут можно почитать о её важности: [1](#), [2](#).

Идеально, всё что вы не модифицируете, должно иметь модификатор *const*.

Интересно, как *const correctness* обеспечивается на низком уровне. Локальные *const*-переменные и аргументы ф-ций не проверяются во время исполнения (только во время компиляции). Но глобальные переменные этого типа располагаются в сегментах данных только для чтения.

Вот пример который упадет, потому что, если будет скомпилирован MSVC для win32, глобальная переменная *a* располагается в сегменте *.rdata*, который только для чтения:

```

const a=123;

void f(int *i)
{
    *i=11; // crash
};

int main()
{
    f(&a);
    return a;
};

```

**Анонимные** (не привязанные к имени переменной) строки в Си имеют тип `const char*`. Вы не можете их модифицировать:

```

#include <string.h>
#include <stdio.h>

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

void alter_string(char *s)
{
    strcpy (s, "Goodbye!");
    printf ("Result: %s\n", s);
};

int main()
{
    alter_string ("Hello, world!\n");
};

```

Это код упадет в Linux (“segmentation fault”) и в Windows, если скомпилирован MinGW.

GCC для Linux располагает все текстовые строки в сегменте данных `.rodata`, который недвусмысленно защищен от записи (“read only data”):

```

$ objdump -s 1

...

Contents of section .rodata:
 400600 01000200 52657375 6c743a20 25730a00  ....Result: %s..
 400610 48656c6c 6f2c2077 6f726c64 210a00    Hello, world!..

```

Когда функция `alter_string()` пытается там писать, срабатывает исключение.

Всё немного иначе в коде сгенерированном MSVC, строки располагаются в сегменте `.data`, у которого нет флага `READONLY`. Оплошность разработчиков MSVC?

```

C:\...>objdump -s 1.exe

...

Contents of section .data:
 40b000 476f6f64 62796521 00000000 52657375  Goodbye!....Resu
 40b010 6c743a20 25730a00 48656c6c 6f2c2077  lt: %s..Hello, w
 40b020 6f726c64 210a0000 00000000 00000000  orld!.....
 40b030 01000000 00000000 c0cb4000 00000000  .....@.....

...

C:\...>objdump -x 1.exe

...

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text          00006d2a  00401000  00401000  00000400  2**2
CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rdata         00002262  00408000  00408000  00007200  2**2
CONTENTS, ALLOC, LOAD, READONLY, DATA

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

2	.data	00000e00	0040b000	0040b000	00009600	2**2
		CONTENTS, ALLOC, LOAD, DATA				
3	.reloc	00000b98	0040e000	0040e000	0000a400	2**2
		CONTENTS, ALLOC, LOAD, READONLY, DATA				

А в MinGW этой ошибки нет, и строки располагаются в сегменте `.rdata`.

### 3.2.1. Пересекающиеся `const`-строки

Тот факт, что *анонимная* Си-строка имеет тип `const` (`??` (стр. `??`)), и тот факт, что выделенные в сегменте констант Си-строки гарантировано неизменяемые (immutable), ведет к интересному следствию: компилятор может использовать определенную часть строки.

Вот простой пример:

```
#include <stdio.h>

int f1()
{
    printf ("world\n");
}

int f2()
{
    printf ("hello world\n");
}

int main()
{
    f1();
    f2();
}
```

Среднестатистический компилятор с Си/Си++(включая MSVC) выделит место для двух строк, но вот что делает GCC 4.8.1:

Листинг 3.1: GCC 4.8.1 + листинг в IDA

f1	proc near
s	= dword ptr -1Ch
	sub esp, 1Ch
	mov [esp+1Ch+s], offset s ; "world\n"
	call _puts
	add esp, 1Ch
	retn
f1	endp
f2	proc near

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

s          = dword ptr -1Ch

          sub     esp, 1Ch
          mov     [esp+1Ch+s], offset aHello ; "hello "
          call   _puts
          add     esp, 1Ch
          retn

f2        endp

aHello    db 'hello '
s         db 'world',0xa,0

```

Действительно, когда мы выводим строку «hello world», эти два слова расположены в памяти вплотную друг к другу и `puts()`, вызываясь из функции `f2()`, вообще не знает, что эти строки разделены. Они и не разделены на самом деле, они разделены только *виртуально*, в нашем листинге.

Когда `puts()` вызывается из `f1()`, он использует строку «world» плюс нулевой байт. `puts()` не знает, что там ещё есть какая-то строка перед этой!

Этот трюк часто используется (по крайней мере в GCC) и может сэкономить немного памяти. Это близко к *string interning*.

Еще один связанный с этим пример находится здесь: ?? (стр. ??).

### 3.3. Пример `strstr()`

Вернемся к тому факту, что GCC иногда использует только часть строки: ?? (стр. ??).

Ф-ция `strstr()` (из стандартной библиотеки Си/Си++) используется для поиска вхождений в строке. Вот что мы сделаем:

```

#include <string.h>
#include <stdio.h>

int main()
{
    char *s="Hello, world!";
    char *w=strstr(s, "world");

    printf ("%p, [%s]\n", s, s);
    printf ("%p, [%s]\n", w, w);
};

```

Вывод:

```

0x8048530, [Hello, world!]
0x8048537, [world!]

```

Разница между адресом оригинальной строки и адресом подстроки, который вернула `strstr()`, это 7. Действительно, строка «Hello, » имеет длину в 7 символов.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Ф-ция `printf()` во время второго вызова не знает о том, что перед переданной строкой имеются еще какие-то символы, и печатает символы с середины оригинальной строки, до конца (который обозначен нулевым байтом).

## 3.4. Конвертирование температуры

Еще один крайне популярный пример из книг по программированию для начинающих, это простейшая программа для конвертирования температуры по Фаренгейту в температуру по Цельсию.

$$C = \frac{5 \cdot (F - 32)}{9}$$

Мы также добавим простейшую обработку ошибок: 1) мы должны проверять правильность ввода пользователем; 2) мы должны проверять результат, не ниже ли он  $-273$  по Цельсию (что, как мы можем помнить из школьных уроков физики, ниже абсолютного нуля).

Функция `exit()` заканчивает программу тут же, без возврата в вызывающую функцию.

### 3.4.1. Целочисленные значения

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int celsius, fahr;
    printf ("Enter temperature in Fahrenheit:\n");
    if (scanf ("%d", &fahr)!=1)
    {
        printf ("Error while parsing your input\n");
        exit(0);
    };

    celsius = 5 * (fahr-32) / 9;

    if (celsius<=-273)
    {
        printf ("Error: incorrect temperature!\n");
        exit(0);
    };
    printf ("Celsius: %d\n", celsius);
};
```

### Оптимизирующий MSVC 2012 x86

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Листинг 3.2: Оптимизирующий MSVC 2012 x86

```

$SG4228 DB      'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4230 DB      '%d', 00H
$SG4231 DB      'Error while parsing your input', 0aH, 00H
$SG4233 DB      'Error: incorrect temperature!', 0aH, 00H
$SG4234 DB      'Celsius: %d', 0aH, 00H

_fahr$ = -4      ; size = 4
_main PROC
    push    ecx
    push    esi
    mov     esi, DWORD PTR __imp__printf
    push    OFFSET $SG4228      ; 'Enter temperature in Fahrenheit:'
    call    esi                ; вызвать printf()
    lea    eax, DWORD PTR _fahr$[esp+12]
    push    eax
    push    OFFSET $SG4230      ; '%d'
    call    DWORD PTR __imp__scanf
    add    esp, 12
    cmp    eax, 1
    je     SHORT $LN2@main
    push    OFFSET $SG4231      ; 'Error while parsing your input'
    call    esi                ; вызвать printf()
    add    esp, 4
    push    0
    call    DWORD PTR __imp__exit

$LN9@main:
$LN2@main:
    mov     eax, DWORD PTR _fahr$[esp+8]
    add    eax, -32             ; ffffffff0H
    lea    ecx, DWORD PTR [eax+eax*4]
    mov     eax, 954437177      ; 38e38e39H
    imul   ecx
    sar    edx, 1
    mov     eax, edx
    shr    eax, 31             ; 0000001fH
    add    eax, edx
    cmp    eax, -273           ; ffffffeefH
    jge    SHORT $LN1@main
    push    OFFSET $SG4233      ; 'Error: incorrect temperature!'
    call    esi                ; вызвать printf()
    add    esp, 4
    push    0
    call    DWORD PTR __imp__exit

$LN10@main:
$LN1@main:
    push    eax
    push    OFFSET $SG4234      ; 'Celsius: %d'
    call    esi                ; вызвать printf()
    add    esp, 8
    ; возврат 0 - по стандарту C99
    xor    eax, eax
    pop    esi

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        pop     ecx
        ret     0
$LN8@main:
_main   ENDP

```

Что мы можем сказать об этом:

- Адрес функции `printf()` в начале загружается в регистр `ESI` так что последующие вызовы `printf()` происходят просто при помощи инструкции `CALL ESI`. Это очень популярная техника компиляторов, может присутствовать, если имеются несколько вызовов одной и той же функции в одном месте, и/или имеется свободный регистр для этого.
- Мы видим инструкцию `ADD EAX, -32` в том месте где от значения должно отняться 32.  $EAX = EAX + (-32)$  эквивалентно  $EAX = EAX - 32$  и как-то компилятор решил использовать `ADD` вместо `SUB`. Может быть оно того стоит, но сказать трудно.
- Инструкция `LEA` используются там, где нужно умножить значение на 5: `lea ecx, DWORD PTR [eax+eax*4]`. Да,  $i + i * 4$  эквивалентно  $i * 5$  и `LEA` работает быстрее чем `IMUL`. Кстати, пара инструкций `SHL EAX, 2 / ADD EAX, EAX` может быть использована здесь вместо `LEA`— некоторые компиляторы так и делают.
- Деление через умножение (?? (стр. ??)) также используется здесь.
- Функция `main()` возвращает 0 хотя `return 0` в конце функции отсутствует. В стандарте C99 [ISO/IEC 9899:TC3 (C C99 standard), (2007)5.1.2.2.3] указано что `main()` будет возвращать 0 в случае отсутствия выражения `return`. Это правило работает только для функции `main()`. И хотя, `MSVC` официально не поддерживает C99, может быть частично и поддерживает?

### Оптимизирующий MSVC 2012 x64

Код почти такой же, хотя мы заметим инструкцию `INT 3` после каждого вызова `exit()`.

```

xor     ecx, ecx
call   QWORD PTR __imp_exit
int     3

```

`INT 3` это точка останова для отладчика.

Известно что функция `exit()` из тех, что никогда не возвращают управление<sup>2</sup>, так что если управление все же возвращается, значит происходит что-то крайне странное, и пришло время запускать отладчик.

### 3.4.2. Числа с плавающей запятой

<sup>2</sup>еще одна популярная из того же ряда это `longjmp()`



```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    double celsius, fahr;
    printf ("Enter temperature in Fahrenheit:\n");
    if (scanf ("%lf", &fahr)!=1)
    {
        printf ("Error while parsing your input\n");
        exit(0);
    };

    celsius = 5 * (fahr-32) / 9;

    if (celsius<=-273)
    {
        printf ("Error: incorrect temperature!\n");
        exit(0);
    };
    printf ("Celsius: %lf\n", celsius);
};

```

MSVC 2010 x86 использует инструкции **FPU!**...

Листинг 3.3: Оптимизирующий MSVC 2010 x86

```

$SG4038 DB      'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4040 DB      '%lf', 00H
$SG4041 DB      'Error while parsing your input', 0aH, 00H
$SG4043 DB      'Error: incorrect temperature!', 0aH, 00H
$SG4044 DB      'Celsius: %lf', 0aH, 00H

__real@c071100000000000 DQ 0c07110000000000r    ; -273
__real@4022000000000000 DQ 0402200000000000r    ; 9
__real@4014000000000000 DQ 0401400000000000r    ; 5
__real@4040000000000000 DQ 0404000000000000r    ; 32

_fahr$ = -8      ; size = 8
_main PROC
    sub     esp, 8
    push   esi
    mov    esi, DWORD PTR __imp__printf
    push   OFFSET $SG4038      ; 'Enter temperature in Fahrenheit:'
    call   esi                 ; вызвать printf()
    lea   eax, DWORD PTR _fahr$[esp+16]
    push   eax
    push   OFFSET $SG4040      ; '%lf'
    call   DWORD PTR __imp__scanf
    add   esp, 12
    cmp   eax, 1
    je    SHORT $LN2@main
    push   OFFSET $SG4041      ; 'Error while parsing your input'

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    call    esi                ; вызвать printf()
    add     esp, 4
    push   0
    call   DWORD PTR __imp__exit
$LN2@main:
    fld    QWORD PTR _fahr$[esp+12]
    fsub   QWORD PTR __real@4040000000000000 ; 32
    fmul   QWORD PTR __real@4014000000000000 ; 5
    fdiv   QWORD PTR __real@4022000000000000 ; 9
    fld    QWORD PTR __real@c071100000000000 ; -273
    fcomp  ST(1)
    fnstsw ax
    test   ah, 65             ; 00000041H
    jne    SHORT $LN1@main
    push   OFFSET $SG4043     ; 'Error: incorrect temperature!'
    fstp   ST(0)
    call   esi                ; вызвать printf()
    add     esp, 4
    push   0
    call   DWORD PTR __imp__exit
$LN1@main:
    sub    esp, 8
    fstp   QWORD PTR [esp]
    push   OFFSET $SG4044     ; 'Celsius: %lf'
    call   esi
    add    esp, 12
    ; возврат 0 - по стандарту C99
    xor    eax, eax
    pop    esi
    add    esp, 8
    ret    0
$LN10@main:
_main    ENDP

```

...но MSVC от года 2012 использует инструкции **SIMD!** вместо этого:

#### Листинг 3.4: Оптимизирующий MSVC 2010 x86

```

$SG4228 DB    'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4230 DB    '%lf', 00H
$SG4231 DB    'Error while parsing your input', 0aH, 00H
$SG4233 DB    'Error: incorrect temperature!', 0aH, 00H
$SG4234 DB    'Celsius: %lf', 0aH, 00H
__real@c071100000000000 DQ 0c07110000000000r ; -273
__real@4040000000000000 DQ 0404000000000000r ; 32
__real@4022000000000000 DQ 0402200000000000r ; 9
__real@4014000000000000 DQ 0401400000000000r ; 5

_fahr$ = -8 ; size = 8
_main PROC
    sub    esp, 8
    push  esi
    mov   esi, DWORD PTR __imp__printf
    push  OFFSET $SG4228 ; 'Enter temperature in Fahrenheit:'

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

call    esi                ; вызвать printf()
lea    eax, DWORD PTR _fahr$[esp+16]
push   eax
push   OFFSET $SG4230     ; '%lf'
call   DWORD PTR __imp__scanf
add    esp, 12
cmp    eax, 1
je     SHORT $LN2@main
push   OFFSET $SG4231     ; 'Error while parsing your input'
call   esi                ; вызвать printf()
add    esp, 4
push   0
call   DWORD PTR __imp__exit
$LN9@main:
$LN2@main:
movsd  xmm1, QWORD PTR _fahr$[esp+12]
subsd  xmm1, QWORD PTR __real@4040000000000000 ; 32
movsd  xmm0, QWORD PTR __real@c071100000000000 ; -273
mulsd  xmm1, QWORD PTR __real@4014000000000000 ; 5
divsd  xmm1, QWORD PTR __real@4022000000000000 ; 9
comisd xmm0, xmm1
jbe    SHORT $LN1@main
push   OFFSET $SG4233     ; 'Error: incorrect temperature!'
call   esi                ; вызвать printf()
add    esp, 4
push   0
call   DWORD PTR __imp__exit
$LN10@main:
$LN1@main:
sub    esp, 8
movsd  QWORD PTR [esp], xmm1
push   OFFSET $SG4234     ; 'Celsius: call esi ; вызвать
printf()
add    esp, 12
; возврат 0 - по стандарту C99
xor    eax, eax
pop    esi
add    esp, 8
ret    0
$LN8@main:
_main  ENDP

```

Конечно, **SIMD!**-инструкции доступны и в x86-режиме, включая те что работают с числами с плавающей запятой. Их использовать в каком-то смысле проще, так что новый компилятор от Microsoft теперь применяет их.

Мы можем также заметить, что значение `-273` загружается в регистр `XMM0` слишком рано. И это нормально, потому что компилятор может генерировать инструкции далеко не в том порядке, в котором они появляются в исходном коде.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

## 3.5. Числа Фибоначчи

Еще один часто используемый пример в учебниках по программированию это рекурсивная функция, генерирующая числа Фибоначчи<sup>3</sup>. Последовательность очень простая: каждое следующее число — это сумма двух предыдущих. Первые два числа — это 0 и 1, или 1 и 1.

Начало последовательности:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181...

### 3.5.1. Пример #1

Реализация проста. Эта программа генерирует последовательность вплоть до 21.

```
#include <stdio.h>

void fib (int a, int b, int limit)
{
    printf ("%d\n", a+b);
    if (a+b > limit)
        return;
    fib (b, a+b, limit);
};

int main()
{
    printf ("0\n1\n1\n");
    fib (1, 1, 20);
};
```

Листинг 3.5: MSVC 2010 x86

```
_a$ = 8           ; size = 4
_b$ = 12         ; size = 4
_limit$ = 16    ; size = 4
_fib PROC
    push        ebp
    mov         ebp, esp
    mov         eax, DWORD PTR _a$[ebp]
    add         eax, DWORD PTR _b$[ebp]
    push        eax
    push        OFFSET $SG2643
    call        DWORD PTR __imp__printf
    add         esp, 8
    mov         ecx, DWORD PTR _a$[ebp]
    add         ecx, DWORD PTR _b$[ebp]
    cmp         ecx, DWORD PTR _limit$[ebp]
    jle         SHORT $LN1@fib
    jmp         SHORT $LN2@fib
```

<sup>3</sup><http://oeis.org/A000045>

```
$LN1@fib:
    mov     edx, DWORD PTR _limit$[ebp]
    push   edx
    mov     eax, DWORD PTR _a$[ebp]
    add     eax, DWORD PTR _b$[ebp]
    push   eax
    mov     ecx, DWORD PTR _b$[ebp]
    push   ecx
    call   _fib
    add     esp, 12
$LN2@fib:
    pop    ebp
    ret    0
_fib    ENDP
_main  PROC
    push   ebp
    mov    ebp, esp
    push   OFFSET $SG2647 ; "0\n1\n1\n"
    call   DWORD PTR __imp__printf
    add    esp, 4
    push   20
    push   1
    push   1
    call   _fib
    add    esp, 12
    xor    eax, eax
    pop    ebp
    ret    0
_main  ENDP
```

Этим мы проиллюстрируем стековые фреймы.

Загрузим пример в OllyDbg и дотрассируем до самого последнего вызова функции f():

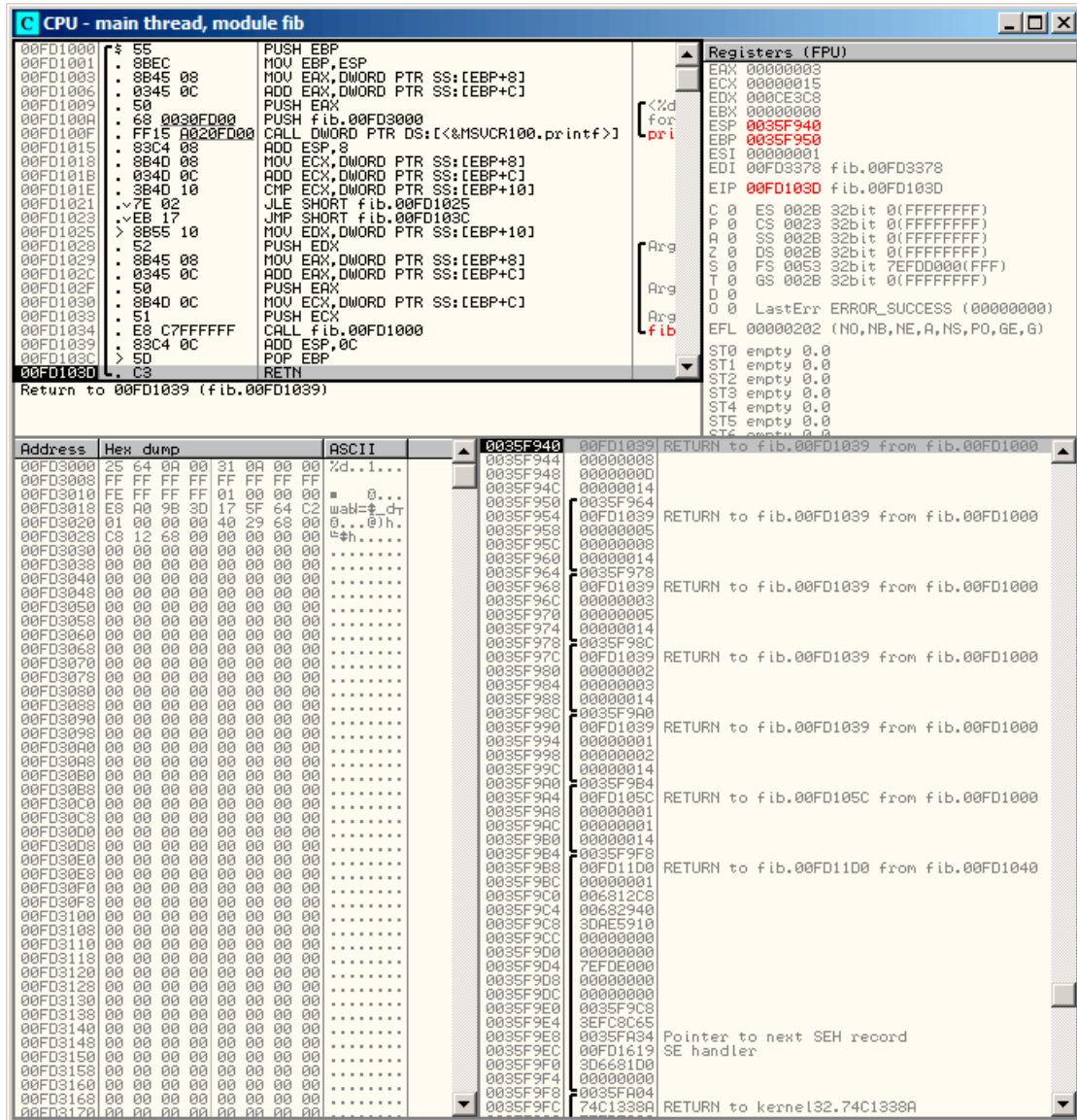


Рис. 3.1: OllyDbg: последний вызов f()

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Исследуем стек более пристально. Комментарии автора книги <sup>4</sup>:

```

0035F940 00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F944 00000008 первый аргумент: a
0035F948 0000000D второй аргумент: b
0035F94C 00000014 третий аргумент: limit
0035F950 /0035F964 сохраненный регистр EBP
0035F954 |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F958 |00000005 первый аргумент: a
0035F95C |00000008 второй аргумент: b
0035F960 |00000014 третий аргумент: limit
0035F964 ]0035F978 сохраненный регистр EBP
0035F968 |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F96C |00000003 первый аргумент: a
0035F970 |00000005 второй аргумент: b
0035F974 |00000014 третий аргумент: limit
0035F978 ]0035F98C сохраненный регистр EBP
0035F97C |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F980 |00000002 первый аргумент: a
0035F984 |00000003 второй аргумент: b
0035F988 |00000014 третий аргумент: limit
0035F98C ]0035F9A0 сохраненный регистр EBP
0035F990 |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F994 |00000001 первый аргумент: a
0035F998 |00000002 второй аргумент: b
0035F99C |00000014 третий аргумент: limit
0035F9A0 ]0035F9B4 сохраненный регистр EBP
0035F9A4 |00FD105C RETURN to fib.00FD105C from fib.00FD1000
0035F9A8 |00000001 первый аргумент: a \
0035F9AC |00000001 второй аргумент: b | подготовлено в main() для f1 ↵
    ↵ ( )
0035F9B0 |00000014 третий аргумент: limit /
0035F9B4 ]0035F9F8 сохраненный регистр EBP
0035F9B8 |00FD11D0 RETURN to fib.00FD11D0 from fib.00FD1040
0035F9BC |00000001 main() первый аргумент: argc \
0035F9C0 |006812C8 main() второй аргумент: argv | подготовлено в CRT для ↵
    ↵ main()
0035F9C4 |00682940 main() третий аргумент: envp /

```

Функция рекурсивная <sup>5</sup>, поэтому стек выглядит как «бутерброд».

Мы видим, что аргумент *limit* всегда один и тот же (0x14 или 20), но аргументы *a* и *b* разные при каждом вызове.

Здесь также адреса **RA!** и сохраненные значения EBP. OllyDbg способна определять EBP-фреймы, так что она тут нарисовала скобки. Значения внутри каждой скобки это **stack frame**, иными словами, место, которое каждая инкарнация функции может использовать для любых своих нужд. Можно сказать, каждая инкарнация функции не должна обращаться к элементам стека за пределами фрейма (не учитывая аргументов функции), хотя это и возможно технически.

<sup>4</sup>Кстати, в OllyDbg можно отметить несколько элементов и скопировать их в клипборд (Ctrl-C). Это было сделано для этого примера

<sup>5</sup>т.е. вызывающая сама себя

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Обычно это так и есть, если только функция не содержит каких-то ошибок. Каждое сохраненное значение EBP это адрес предыдущего [stack frame](#): это причина, почему некоторые отладчики могут легко делить стек на фреймы и выводить аргументы каждой функции.

Как видно, каждая инкарнация функции готовит аргументы для следующего вызова функции.

В самом конце мы видим 3 аргумента функции `main()`. `argc` равен 1 (да, действительно, ведь мы запустили эту программу без аргументов в командной строке).

Очень легко привести к переполнению стека: просто удалите (или закомментируйте) проверку предела и процесс упадет с исключением `0xC00000FD` (переполнение стека.)

### 3.5.2. Пример #2

В моей функции есть некая избыточность, так что добавим переменную `next` и заменим на нее все «`a+b`»:

```
#include <stdio.h>

void fib (int a, int b, int limit)
{
    int next=a+b;
    printf ("%d\n", next);
    if (next > limit)
        return;
    fib (b, next, limit);
};

int main()
{
    printf ("0\n1\n1\n");
    fib (1, 1, 20);
};
```

Это результат работы неоптимизирующего MSVC, поэтому переменная `next` действительно находится в локальном стеке:

Листинг 3.6: MSVC 2010 x86

```
_next$ = -4      ; size = 4
_a$ = 8         ; size = 4
_b$ = 12        ; size = 4
_limit$ = 16    ; size = 4
_fib          PROC
    push      ebp
    mov      ebp, esp
    push      ecx
    mov      eax, DWORD PTR _a$[ebp]
    add      eax, DWORD PTR _b$[ebp]
    mov      DWORD PTR _next$[ebp], eax
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

    mov     ecx, DWORD PTR _next$[ebp]
    push   ecx
    push   OFFSET $SG2751 ; '%d'
    call   DWORD PTR __imp__printf
    add    esp, 8
    mov    edx, DWORD PTR _next$[ebp]
    cmp    edx, DWORD PTR _limit$[ebp]
    jle    SHORT $LN1@fib
    jmp    SHORT $LN2@fib
$LN1@fib:
    mov    eax, DWORD PTR _limit$[ebp]
    push  eax
    mov    ecx, DWORD PTR _next$[ebp]
    push  ecx
    mov    edx, DWORD PTR _b$[ebp]
    push  edx
    call  _fib
    add    esp, 12
$LN2@fib:
    mov    esp, ebp
    pop    ebp
    ret    0
_fib     ENDP

_main   PROC
    push  ebp
    mov   ebp, esp
    push  OFFSET $SG2753 ; "0\n1\n1\n"
    call  DWORD PTR __imp__printf
    add   esp, 4
    push  20
    push  1
    push  1
    call  _fib
    add   esp, 12
    xor   eax, eax
    pop   ebp
    ret   0
_main   ENDP

```

Загрузим OllyDbg снова:

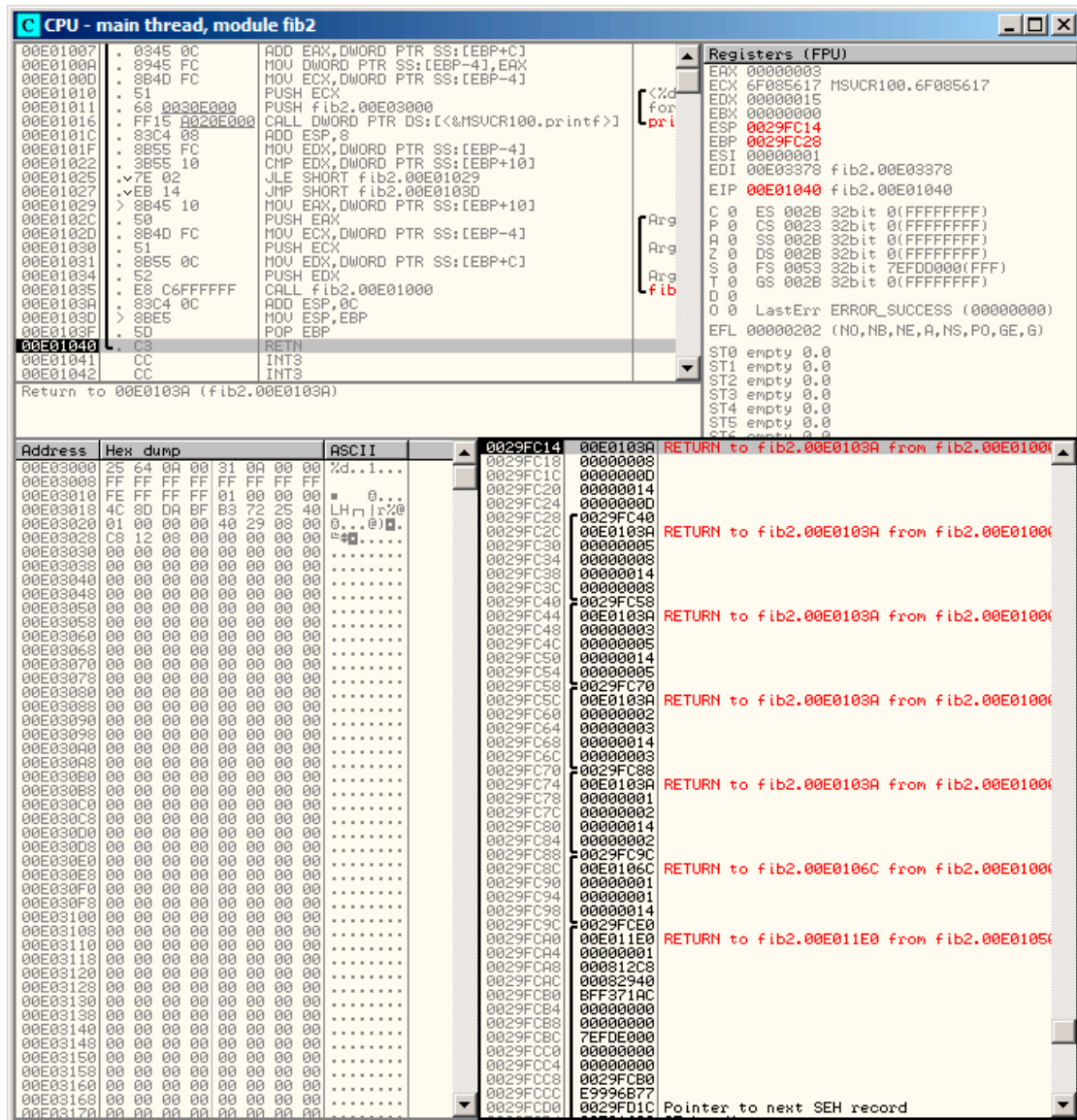


Рис. 3.2: OllyDbg: последний вызов `f()`

Теперь переменная `next` присутствует в каждом фрейме.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Рассмотрим стек более пристально. Автор и здесь добавил туда своих комментариев:

```

0029FC14  00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC18  00000008 первый аргумент: a
0029FC1C  0000000D второй аргумент: b
0029FC20  00000014 третий аргумент: limit
0029FC24  0000000D переменная "next"
0029FC28  /0029FC40 сохраненный регистр EBP
0029FC2C  |00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC30  |00000005 первый аргумент: a
0029FC34  |00000008 второй аргумент: b
0029FC38  |00000014 третий аргумент: limit
0029FC3C  |00000008 переменная "next"
0029FC40  |0029FC58 сохраненный регистр EBP
0029FC44  |00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC48  |00000003 первый аргумент: a
0029FC4C  |00000005 второй аргумент: b
0029FC50  |00000014 третий аргумент: limit
0029FC54  |00000005 переменная "next"
0029FC58  |0029FC70 сохраненный регистр EBP
0029FC5C  |00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC60  |00000002 первый аргумент: a
0029FC64  |00000003 второй аргумент: b
0029FC68  |00000014 третий аргумент: limit
0029FC6C  |00000003 переменная "next"
0029FC70  |0029FC88 сохраненный регистр EBP
0029FC74  |00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC78  |00000001 первый аргумент: a      \
0029FC7C  |00000002 второй аргумент: b    | подготовлено в f1() для ↵
    ↵ следующего вызова f1()
0029FC80  |00000014 третий аргумент: limit /
0029FC84  |00000002 переменная "next"
0029FC88  |0029FC9C сохраненный регистр EBP
0029FC8C  |00E0106C RETURN to fib2.00E0106C from fib2.00E01000
0029FC90  |00000001 первый аргумент: a      \
0029FC94  |00000001 второй аргумент: b    | подготовлено в main() для f1↵
    ↵ ()
0029FC98  |00000014 третий аргумент: limit /
0029FC9C  |0029FCE0 сохраненный регистр EBP
0029FCA0  |00E011E0 RETURN to fib2.00E011E0 from fib2.00E01050
0029FCA4  |00000001 main() первый аргумент: argc \
0029FCA8  |000812C8 main() второй аргумент: argv | подготовлено в CRT для ↵
    ↵ main()
0029FCAC  |00082940 main() третий аргумент: envp /

```

Значение переменной *next* вычисляется в каждой инкарнации функции, затем передается аргумент *b* в следующую инкарнацию.

### 3.5.3. Итог

Рекурсивные функции эстетически красивы, но технически могут ухудшать производительность из-за активного использования стека. Тот, кто пишет кри-

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

тические к времени исполнения участки кода, наверное, должен избегать применения там рекурсии.

Например, однажды автор этих строк написал функцию для поиска нужного узла в двоичном дереве. Рекурсивно она выглядела очень красиво, но из-за того, что при каждом вызове тратилось время на эпилог и пролог, все это работало в несколько раз медленнее чем та же функция, но без рекурсии.

Кстати, поэтому некоторые компиляторы функциональных **PL!**<sup>6</sup> (где рекурсия активно применяется) используют **хвостовую рекурсию**. Хвостовая рекурсия, это когда ф-ция имеет только один вызов самой себя, в самом конце, например:

Листинг 3.7: Scheme, пример взят из Wikipedia

```
;; factorial : number -> number
;; to calculate the product of all positive
;; integers less than or equal to n.
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

Хвостовая рекурсия важна, потому что компилятор может легко переработать такой код в итеративный, чтобы избавиться от рекурсии.

## 3.6. Пример вычисления CRC32

Это распространенный табличный способ вычисления хеша алгоритмом CRC32<sup>7</sup>.

```
/* By Bob Jenkins, (c) 2006, Public Domain */

#include <stdio.h>
#include <stddef.h>
#include <string.h>

typedef unsigned long ub4;
typedef unsigned char ub1;

static const ub4 crctab[256] = {
  0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419,
  0x706af48f, 0xe963a535, 0x9e6495a3, 0x0edb8832, 0x79dcb8a4,
  0xe0d5e91e, 0x97d2d988, 0x09b64c2b, 0x7eb17cbd, 0xe7b82d07,
  0x90bf1d91, 0x1db71064, 0x6ab020f2, 0xf3b97148, 0x84be41de,
  0x1dad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7, 0x136c9856,
  0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0x63066cd9,
  0xfa0f3d63, 0x8d080df5, 0x3b6e20c8, 0x4c69105e, 0xd56041e4,
  0xa2677172, 0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b,
  0x35b5a8fa, 0x42b2986c, 0xdbbbc9d6, 0xacbcf940, 0x32d86ce3,
  0x45df5c75, 0xdcd60dcf, 0xabd13d59, 0x26d930ac, 0x51de003a,
```

<sup>6</sup>LISP, Python, Lua, etc.

<sup>7</sup>Исходник взят тут: <http://burtleburtle.net/bob/c/crc.c>

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
printf("0x%.8lx, ", j);
if (i%6 == 5) printf("\n");
}
}

/* the hash function */
ub4 crc(const void *key, ub4 len, ub4 hash)
{
    ub4 i;
    const ub1 *k = key;
    for (hash=len, i=0; i<len; ++i)
        hash = (hash >> 8) ^ crctab[(hash & 0xff) ^ k[i]];
    return hash;
}

/* To use, try "gcc -O crc.c -o crc; crc < crc.c" */
int main()
{
    char s[1000];
    while (gets(s)) printf("%.8lx\n", crc(s, strlen(s), 0));
    return 0;
}

```

Нас интересует функция `crc()`. Кстати, обратите внимание на два инициализатора в выражении `for()`: `hash=len`, `i=0`. Стандарт Си/Си++, конечно, допускает это. А в итоговом коде, вместо одной операции инициализации цикла, будет две.

Компилируем в MSVC с оптимизацией (`/Ox`). Для краткости, я приведу только функцию `crc()`, с некоторыми комментариями.

```

_key$ = 8           ; size = 4
_len$ = 12          ; size = 4
_hash$ = 16         ; size = 4
_crc PROC
    mov     edx, DWORD PTR _len$[esp-4]
    xor     ecx, ecx ; i будет лежать в регистре ECX
    mov     eax, edx
    test    edx, edx
    jbe    SHORT $LN1@crc
    push    ebx
    push    esi
    mov     esi, DWORD PTR _key$[esp+4] ; ESI = key
    push    edi
$LL3@crc:
; работаем с байтами используя 32-битные регистры. в EDI положим байт с
; адреса key+i

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

movzx edi, BYTE PTR [ecx+esi]
mov ebx, eax ; EBX = (hash = len)
and ebx, 255 ; EBX = hash & 0xff

; XOR EDI, EBX (EDI=EDI^EBX)
; эта операция задействует все 32 бита каждого регистра
; но остальные биты (8-31) будут обнулены всегда, так что все ОК
; они обнулены потому что для EDI это было сделано инструкцией MOVZX выше
; а старшие биты EBX были сброшены инструкцией AND EBX, 255 (255 = 0xff)

xor edi, ebx

; EAX=EAX>>8; образовавшиеся из ниоткуда биты в результате (биты 24-31) будут
; заполнены нулями
shr eax, 8

;
EAX=EAX^crctab[EDI*4] - выбираем элемент из таблицы crctab[] под номером EDI
xor eax, DWORD PTR _crctab[edi*4]
inc ecx ; i++
cmp ecx, edx ; i<len ?
jb SHORT $LL3@crc ; да
pop edi
pop esi
pop ebx
$LN1@crc:
ret 0
_crc ENDP

```

Попробуем то же самое в GCC 4.4.1 с опцией -O3:

```

crc public crc
proc near

key = dword ptr 8
hash = dword ptr 0Ch

push ebp
xor edx, edx
mov ebp, esp
push esi
mov esi, [ebp+key]
push ebx
mov ebx, [ebp+hash]
test ebx, ebx
mov eax, ebx
jz short loc_80484D3
pop ; выравнивание
lea esi, [esi+0] ; выравнивание; работает как NOP (ESI не
; меняется здесь)

loc_80484B8:
mov ecx, eax ; сохранить предыдущее состояние хеша в
; ECX

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

xor     al, [esi+edx]; AL=*(key+i)
add     edx, 1      ; i++
shr     ecx, 8      ; ECX=hash>>8
movzx   eax, al     ; EAX=*(key+i)
mov     eax, dword ptr ds:crctab[eax*4] ; EAX=crctab[EAX]
xor     eax, ecx    ; hash=EAX^ECX
cmp     ebx, edx
ja      short loc_80484B8

loc_80484D3:
pop     ebx
pop     esi
pop     ebp
retn
crc     endp

```

GCC немного выровнял начало тела цикла по 8-байтной границе, для этого добавил

`NOP` и `lea esi, [esi+0]` (что тоже *холостая операция*). Подробнее об этом смотрите в разделе о `prad` (?? (стр. ??)).

### 3.7. Пример вычисления адреса сети

Как мы знаем, TCP/IP-адрес (IPv4) состоит из четырех чисел в пределах 0...255, т.е. 4 байта.

4 байта легко помещаются в 32-битную переменную, так что адрес хоста в IPv4, сетевая маска или адрес сети могут быть 32-битными числами.

С точки зрения пользователя, маска сети определяется четырьмя числами в формате вроде

255.255.255.0, но сетевые инженеры (сисадмины) используют более компактную нотацию (**CIDR!**<sup>8</sup>), вроде «/8», «/16», и т. д.

Эта нотация просто определяет количество бит в сетевой маске, начиная с **MSB!**.

<sup>8</sup>**CIDR!**



Маска	Хосты	Свободно	Сетевая маска	В шест. виде	
/30	4	2	255.255.255.252	0xffffffffc	
/29	8	6	255.255.255.248	0xffffffff8	
/28	16	14	255.255.255.240	0xffffffff0	
/27	32	30	255.255.255.224	0xffffffe0	
/26	64	62	255.255.255.192	0xffffffc0	
/24	256	254	255.255.255.0	0xfffff000	сеть класса С
/23	512	510	255.255.254.0	0xffffe000	
/22	1024	1022	255.255.252.0	0xffffc000	
/21	2048	2046	255.255.248.0	0xffff8000	
/20	4096	4094	255.255.240.0	0xffff0000	
/19	8192	8190	255.255.224.0	0xffffe000	
/18	16384	16382	255.255.192.0	0xfffc0000	
/17	32768	32766	255.255.128.0	0xffff8000	
/16	65536	65534	255.255.0.0	0xffff0000	сеть класса В
/8	16777216	16777214	255.0.0.0	0xff000000	сеть класса А

Вот простой пример, вычисляющий адрес сети используя сетевую маску и адрес хоста.

```
#include <stdio.h>
#include <stdint.h>

uint32_t form_IP (uint8_t ip1, uint8_t ip2, uint8_t ip3, uint8_t ip4)
{
    return (ip1<<24) | (ip2<<16) | (ip3<<8) | ip4;
};

void print_as_IP (uint32_t a)
{
    printf ("%d.%d.%d.%d\n",
            (a>>24)&0xFF,
            (a>>16)&0xFF,
            (a>>8)&0xFF,
            (a)&0xFF);
};

// bit=31..0
uint32_t set_bit (uint32_t input, int bit)
{
    return input=input|(1<<bit);
};

uint32_t form_netmask (uint8_t netmask_bits)
{
    uint32_t netmask=0;
    uint8_t i;

    for (i=0; i<netmask_bits; i++)
        netmask=set_bit(netmask, 31-i);

    return netmask;
};
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

void calc_network_address (uint8_t ip1, uint8_t ip2, uint8_t ip3, uint8_t ip4, uint8_t netmask_bits)
{
    uint32_t netmask=form_netmask(netmask_bits);
    uint32_t ip=form_IP(ip1, ip2, ip3, ip4);
    uint32_t netw_adr;

    printf ("netmask=");
    print_as_IP (netmask);

    netw_adr=ip&netmask;

    printf ("network address=");
    print_as_IP (netw_adr);
};

int main()
{
    calc_network_address (10, 1, 2, 4, 24);    // 10.1.2.4, /24
    calc_network_address (10, 1, 2, 4, 8);     // 10.1.2.4, /8
    calc_network_address (10, 1, 2, 4, 25);    // 10.1.2.4, /25
    calc_network_address (10, 1, 2, 64, 26);   // 10.1.2.4, /26
};

```

### 3.7.1. calc\_network\_address()

Функция calc\_network\_address() самая простая:

она просто умножает (логически, используя AND) адрес хоста на сетевую маску, в итоге давая адрес сети.

Листинг 3.8: Оптимизирующий MSVC 2012 /Ob0

```

1  _ip1$ = 8           ; size = 1
2  _ip2$ = 12          ; size = 1
3  _ip3$ = 16          ; size = 1
4  _ip4$ = 20          ; size = 1
5  _netmask_bits$ = 24 ; size = 1
6  _calc_network_address PROC
7      push     edi
8      push     DWORD PTR _netmask_bits$[esp]
9      call    _form_netmask
10     push     OFFSET $SG3045 ; 'netmask='
11     mov     edi, eax
12     call    DWORD PTR __imp__printf
13     push     edi
14     call    _print_as_IP
15     push     OFFSET $SG3046 ; 'network address='
16     call    DWORD PTR __imp__printf
17     push     DWORD PTR _ip4$[esp+16]
18     push     DWORD PTR _ip3$[esp+20]
19     push     DWORD PTR _ip2$[esp+24]
20     push     DWORD PTR _ip1$[esp+28]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

21     call    _form_IP
22     and     eax, edi           ; network address = host address & netmask
23     push   eax
24     call    _print_as_IP
25     add     esp, 36
26     pop     edi
27     ret     0
28 _calc_network_address ENDP

```

На строке 22 мы видим самую важную инструкцию AND— так вычисляется адрес сети.

### 3.7.2. form\_IP()

Функция form\_IP() просто собирает все 4 байта в одно 32-битное значение.

Вот как это обычно происходит:

- Выделите переменную для возвращаемого значения. Обнулите её.
- Возьмите четвертый (самый младший) байт, сложите его (логически, инструкцией OR) с возвращаемым значением. Оно содержит теперь 4-й байт.
- Возьмите третий байт, сдвиньте его на 8 бит влево. Получится значение в виде 0x0000bb00, где bb это третий байт. Сложите итоговое значение (логически, инструкцией OR) с возвращаемым значением. Возвращаемое значение пока что содержит 0x000000aa, так что логическое сложение в итоге выдаст значение вида 0x0000bbaa.
- Возьмите второй байт, сдвиньте его на 16 бит влево. Вы получите значение вида 0x00cc0000, где cc это второй байт. Сложите (логически) результат и возвращаемое значение. Выходное значение содержит пока что 0x0000bbaa, так что логическое сложение в итоге выдаст значение вида 0x00ccbbaa.
- Возьмите первый байт, сдвиньте его на 24 бита влево. Вы получите значение вида 0xdd000000, где dd это первый байт. Сложите (логически) результат и выходное значение. Выходное значение содержит пока что 0x00ccbbaa, так что сложение выдаст в итоге значение вида 0xddccbbaa.

И вот как работает неоптимизирующий MSVC 2012:

Листинг 3.9: Неоптимизирующий MSVC 2012

```

; определим ip1 как "dd", ip2 как "cc", ip3 как "bb", ip4 как "aa".
_ip1$ = 8           ; size = 1
_ip2$ = 12          ; size = 1
_ip3$ = 16          ; size = 1
_ip4$ = 20          ; size = 1
_form_IP PROC
    push    ebp
    mov     ebp, esp
    movzx   eax, BYTE PTR _ip1$[ebp]
    ; EAX=000000dd

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

shl    eax, 24
; EAX=dd000000
movzx  ecx, BYTE PTR _ip2$[ebp]
; ECX=000000cc
shl    ecx, 16
; ECX=00cc0000
or     eax, ecx
; EAX=ddcc0000
movzx  edx, BYTE PTR _ip3$[ebp]
; EDX=000000bb
shl    edx, 8
; EDX=0000bb00
or     eax, edx
; EAX=ddccb000
movzx  ecx, BYTE PTR _ip4$[ebp]
; ECX=000000aa
or     eax, ecx
; EAX=ddccb00a
pop    ebp
ret    0
_form_IP ENDP

```

Хотя, порядок операций другой, но, конечно, порядок роли не играет.  
 Оптимизирующий MSVC 2012 делает то же самое, но немного иначе:

Листинг 3.10: Оптимизирующий MSVC 2012 /Ob0

```

; определим ip1 как "dd", ip2 как "cc", ip3 как "bb", ip4 как "aa".
_ip1$ = 8      ; size = 1
_ip2$ = 12     ; size = 1
_ip3$ = 16     ; size = 1
_ip4$ = 20     ; size = 1
_form_IP PROC
movzx  eax, BYTE PTR _ip1$[esp-4]
; EAX=000000dd
movzx  ecx, BYTE PTR _ip2$[esp-4]
; ECX=000000cc
shl    eax, 8
; EAX=0000dd00
or     eax, ecx
; EAX=0000ddcc
movzx  ecx, BYTE PTR _ip3$[esp-4]
; ECX=000000bb
shl    eax, 8
; EAX=00ddcc00
or     eax, ecx
; EAX=00ddccb0
movzx  ecx, BYTE PTR _ip4$[esp-4]
; ECX=000000aa
shl    eax, 8
; EAX=ddccb000
or     eax, ecx
; EAX=ddccb00a

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения,  
 пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    ret     0
_form_IP ENDP

```

Можно сказать, что каждый байт записывается в младшие 8 бит возвращаемого значения, и затем возвращаемое значение сдвигается на один байт влево на каждом шаге.

Повторять 4 раза, для каждого байта.

Вот и всё! К сожалению, наверно, нет способа делать это иначе. Не существует более-менее популярных **CPU!** или **ISA!**, где имеется инструкция для сборки значения из бит или байт. Обычно всё это делает сдвигами бит и логическим сложением (OR).

### 3.7.3. print\_as\_IP()

print\_as\_IP() делает наоборот: расщепляет 32-битное значение на 4 байта.

Расщепление работает немного проще: просто сдвигайте входное значение на 24, 16, 8 или 0 бит, берите биты с нулевого по седьмой (младший байт), вот и всё:

Листинг 3.11: Неоптимизирующий MSVC 2012

```

_a$ = 8 ; size = 4
_print_as_IP PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    ; EAX=ddccbbaa
    and     eax, 255
    ; EAX=000000aa
    push    eax
    mov     ecx, DWORD PTR _a$[ebp]
    ; ECX=ddccbbaa
    shr     ecx, 8
    ; ECX=00ddccbb
    and     ecx, 255
    ; ECX=000000bb
    push    ecx
    mov     edx, DWORD PTR _a$[ebp]
    ; EDX=ddccbbaa
    shr     edx, 16
    ; EDX=0000ddcc
    and     edx, 255
    ; EDX=000000cc
    push    edx
    mov     eax, DWORD PTR _a$[ebp]
    ; EAX=ddccbbaa
    shr     eax, 24
    ; EAX=000000dd
    and     eax, 255 ; возможно, избыточная инструкция
    ; EAX=000000dd
    push    eax

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

push    OFFSET $SG2973 ; '%d.%d.%d.%d'
call    DWORD PTR __imp__printf
add     esp, 20
pop     ebp
ret     0
_print_as_IP ENDP

```

Оптимизирующий MSVC 2012 делает почти всё то же самое, только без ненужных перезагрузок входного значения:

Листинг 3.12: Оптимизирующий MSVC 2012 /Ob0

```

_a$ = 8 ; size = 4
_print_as_IP PROC
mov     ecx, DWORD PTR _a$[esp-4]
; ECX=ddccbbaa
movzx   eax, cl
; EAX=000000aa
push    eax
mov     eax, ecx
; EAX=ddccbbaa
shr     eax, 8
; EAX=00ddccbb
and     eax, 255
; EAX=000000bb
push    eax
mov     eax, ecx
; EAX=ddccbbaa
shr     eax, 16
; EAX=0000ddcc
and     eax, 255
; EAX=000000cc
push    eax
; ECX=ddccbbaa
shr     ecx, 24
; ECX=000000dd
push    ecx
push    OFFSET $SG3020 ; '%d.%d.%d.%d'
call    DWORD PTR __imp__printf
add     esp, 20
ret     0
_print_as_IP ENDP

```

### 3.7.4. form\_netmask() и set\_bit()

form\_netmask() делает сетевую маску из **CIDR!**-нотации.

Конечно, было бы куда эффективнее использовать здесь какую-то уже готовую таблицу, но мы рассматриваем это именно так, сознательно, для демонстрации битовых сдвигов. Мы также сделаем отдельную функцию set\_bit().

Не очень хорошая идея выделять отдельную функцию для такой примитивной операции, но так будет проще понять, как это всё работает.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Листинг 3.13: Оптимизирующий MSVC 2012 /Ob0

```

_input$ = 8          ; size = 4
_bit$ = 12          ; size = 4
_set_bit PROC
    mov     ecx, DWORD PTR _bit$[esp-4]
    mov     eax, 1
    shl     eax, cl
    or      eax, DWORD PTR _input$[esp-4]
    ret     0
_set_bit ENDP

_netmask_bits$ = 8   ; size = 1
_form_netmask PROC
    push    ebx
    push    esi
    movzx   esi, BYTE PTR _netmask_bits$[esp+4]
    xor     ecx, ecx
    xor     bl, bl
    test    esi, esi
    jle     SHORT $LN9@form_netma
    xor     edx, edx
$LL3@form_netma:
    mov     eax, 31
    sub     eax, edx
    push    eax
    push    ecx
    call    _set_bit
    inc     bl
    movzx   edx, bl
    add     esp, 8
    mov     ecx, eax
    cmp     edx, esi
    jl      SHORT $LL3@form_netma
$LN9@form_netma:
    pop     esi
    mov     eax, ecx
    pop     ebx
    ret     0
_form_netmask ENDP

```

set\_bit() примитивна: просто сдвигает единицу на нужное количество бит, затем складывает (логически) с входным значением «input». form\_netmask() имеет цикл: он выставит столько бит (начиная с **MSB!**), сколько передано в аргументе netmask\_bits.

### 3.7.5. Итог

Вот и всё! Мы запускаем и видим:

```

netmask=255.255.255.0
network address=10.1.2.0
netmask=255.0.0.0

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
network address=10.0.0.0
netmask=255.255.255.128
network address=10.1.2.0
netmask=255.255.255.192
network address=10.1.2.64
```

## 3.8. Циклы: несколько итераторов

Часто, у цикла только один итератор, но в итоговом коде их может быть несколько.

Вот очень простой пример:

```
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
    size_t i;

    // копировать из одного массива в другой по какой-то странной схеме
    for (i=0; i<cnt; i++)
        a1[i*3]=a2[i*7];
};
```

Здесь два умножения на каждой итерации, а это дорогая операция.

Сможем ли мы оптимизировать это как-то? Да, если мы заметим, что индексы обоих массивов перескакивают на места, рассчитать которые мы можем легко и без умножения.

### 3.8.1. Три итератора

Листинг 3.14: Оптимизирующий MSVC 2013 x64

```
f      PROC
; RCX=a1
; RDX=a2
; R8=cnt
        test    r8, r8          ; cnt==0? тогда выйти
        je     SHORT $LN1@f
        npad   11
$LL3@f:
        mov     eax, DWORD PTR [rdx]
        lea    rcx, QWORD PTR [rcx+12]
        lea    rdx, QWORD PTR [rdx+28]
        mov     DWORD PTR [rcx-12], eax
        dec    r8
        jne    SHORT $LL3@f
$LN1@f:
        ret     0
f      ENDP
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



Теперь здесь три итератора: переменная *cnt* и два индекса, они увеличиваются на 12 и 28 на каждой итерации, указывая на новые элементы массивов.

Мы можем переписать этот код на Си/Си++:

```
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
    size_t i;
    size_t idx1=0; idx2=0;

    // копировать из одного массива в другой по какой-то странной схеме
    for (i=0; i<cnt; i++)
    {
        a1[idx1]=a2[idx2];
        idx1+=3;
        idx2+=7;
    };
};
```

Так что, ценой модификации трех итераторов на каждой итерации вместо одного, мы избавлены от двух операций умножения.

### 3.8.2. Два итератора

GCC 4.9 сделал еще больше, оставив только 2 итератора:

Листинг 3.15: Оптимизирующий GCC 4.9 x64

```
; RDI=a1
; RSI=a2
; RDX=cnt
f:
    test    rdx, rdx ; cnt==0? тогда выйти
    je     .L1
; вычислить адрес последнего элемента в "a2" и оставить его в RDX
    lea    rax, [0+rdx*4]
; RAX=RDX*4=cnt*4
    sal    rdx, 5
; RDX=RDX<<5=cnt*32
    sub    rdx, rax
; RDX=RDX-RAX=cnt*32-cnt*4=cnt*28
    add    rdx, rsi
; RDX=RDX+RSI=a2+cnt*28
.L3:
    mov    eax, DWORD PTR [rsi]
    add    rsi, 28
    add    rdi, 12
    mov    DWORD PTR [rdi-12], eax
    cmp    rsi, rdx
    jne    .L3
.L1:
    rep   ret
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мой адреса](#). Спасибо!

Здесь больше нет переменной-счетчика: GCC рассудил, что она не нужна.

Последний элемент массива *a2* вычисляется перед началом цикла (а это просто:  $cnt * 7$ ), и при помощи этого цикл останавливается: просто исполняйте его пока второй индекс не сравняется с предварительно вычисленным значением.

Об умножении используя сдвиги/сложения/вычитания, читайте здесь:

?? (стр. ??).

Этот код можно переписать на Си/Си++ вот так:

```
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
    size_t idx1=0; idx2=0;
    size_t last_idx2=cnt*7;

    // копировать из одного массива в другой по какой-то странной схеме
    for (;;)
    {
        a1[idx1]=a2[idx2];
        idx1+=3;
        idx2+=7;
        if (idx2==last_idx2)
            break;
    };
};
```

GCC (Linaro) 4.9 для ARM64 делает тоже самое, только предварительно вычисляет последний индекс массива *a1* вместо *a2*, а это, конечно, имеет тот же эффект:

Листинг 3.16: Оптимизирующий GCC (Linaro) 4.9 ARM64

```
; X0=a1
; X1=a2
; X2=cnt
f:
    cbz    x2, .L1      ; cnt==0? тогда выйти
; вычислить последний элемент массива "a1"
    add    x2, x2, x2, lsl 1
; X2=X2+X2<<1=X2+X2*2=X2*3
    mov    x3, 0
    lsl    x2, x2, 2
; X2=X2<<2=X2*4=X2*3*4=X2*12
.L3:
    ldr    w4, [x1],28 ; загрузить по адресу в X1, прибавить 28 к X1
    (пост-инкремент)
    str    w4, [x0,x3] ; записать по адресу в X0+X3=a1+X3
    add    x3, x3, 12 ; сдвинуть X3
    cmp    x3, x2      ; конец?
    bne    .L3
.L1:
    ret
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

GCC 4.4.5 для MIPS делает то же самое:

Листинг 3.17: Оптимизирующий GCC 4.4.5 для MIPS (IDA)

```

; $a0=a1
; $a1=a2
; $a2=cnt
f:
; переход на код проверки в цикле:
    beqz    $a2, locret_24
; инициализировать счетчик (i) в 0:
    move    $v0, $zero ; branch delay slot, NOP

loc_8:
; загрузить 32-битное слово в $a1
    lw     $a3, 0($a1)
; инкремент счетчика (i):
    addiu  $v0, 1
; проверка на конец (сравнить "i" в $v0 и "cnt" в $a2):
    sltu   $v1, $v0, $a2
; сохранить 32-битное слово в $a0:
    sw     $a3, 0($a0)
; прибавить 0x1C (28) к $a1 на каждой итерации:
    addiu  $a1, 0x1C
; перейти на тело цикла, если i<cnt:
    bnez   $v1, loc_8
; прибавить 0xC (12) к $a0 на каждой итерации:
    addiu  $a0, 0xC ; branch delay slot

locret_24:
    jr     $ra
    or     $at, $zero ; branch delay slot, NOP

```

### 3.8.3. Случай Intel C++ 2011

Оптимизации компилятора могут быть очень странными, но, тем не менее, корректными.

Вот что делает Intel C++ 2011:

Листинг 3.18: Оптимизирующий Intel C++ 2011 x64

```

f      PROC
; parameter 1: rcx = a1
; parameter 2: rdx = a2
; parameter 3: r8 = cnt
.B1.1::
    test    r8, r8
    jbe     exit

.B1.2::
    cmp     r8, 6
    jbe     just_copy

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.B1.3::
    cmp     rcx, rdx
    jbe     .B1.5

.B1.4::
    mov     r10, r8
    mov     r9, rcx
    shl     r10, 5
    lea     rax, QWORD PTR [r8*4]
    sub     r9, rdx
    sub     r10, rax
    cmp     r9, r10
    jge     just_copy2

.B1.5::
    cmp     rdx, rcx
    jbe     just_copy

.B1.6::
    mov     r9, rdx
    lea     rax, QWORD PTR [r8*8]
    sub     r9, rcx
    lea     r10, QWORD PTR [rax+r8*4]
    cmp     r9, r10
    jl     just_copy

just_copy2::
; R8 = cnt
; RDX = a2
; RCX = a1
    xor     r10d, r10d
    xor     r9d, r9d
    xor     eax, eax

.B1.8::
    mov     r11d, DWORD PTR [rax+rdx]
    inc     r10
    mov     DWORD PTR [r9+rcx], r11d
    add     r9, 12
    add     rax, 28
    cmp     r10, r8
    jb     .B1.8
    jmp     exit

just_copy::
; R8 = cnt
; RDX = a2
; RCX = a1
    xor     r10d, r10d
    xor     r9d, r9d
    xor     eax, eax

.B1.11::

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov     r11d, DWORD PTR [rax+rdx]
inc     r10
mov     DWORD PTR [r9+rcx], r11d
add     r9, 12
add     rax, 28
cmp     r10, r8
jb     .B1.11

exit::
ret

```

В начале, принимаются какие-то решения, затем исполняется одна из процедур.

Видимо, это проверка, не пересекаются ли массивы.

Это хорошо известный способ оптимизации процедур копирования блоков в памяти.

Но копирующие процедуры одинаковые! Видимо, это ошибка оптимизатора Intel C++, который, тем не менее, генерирует работоспособный код.

Мы намеренно изучаем примеры такого кода в этой книге чтобы читатель мог понимать, что результаты работы компилятором иногда бывают крайне странными, но корректными, потому что когда компилятор тестировали, тесты прошли нормально.

### 3.9. Duff's device

Duff's device это развернутый цикл с возможностью перехода в середину цикла. Развернутый цикл реализован используя fallthrough-выражение `switch()`. Мы будем использовать здесь упрощенную версию кода Тома Даффа. Скажем, нам нужно написать функцию, очищающую регион в памяти. Кто-то может подумать о простом цикле, очищающем байт за байтом. Это, очевидно, медленно, так как все современные компьютеры имеют намного более широкую шину памяти. Так что более правильный способ — это очищать регион в памяти блоками по 4 или 8 байт. Так как мы будем работать с 64-битным примером, мы будем очищать память блоками по 8 байт.

Пока всё хорошо. Но что насчет хвоста? Функция очистки памяти будет также вызываться и для блоков с длиной не кратной 8.

Вот алгоритм:

- вычислить количество 8-байтных блоков, очистить их используя 8-байтный (64-битный) доступ к памяти;
- вычислить размер хвоста, очистить его используя 1-байтный доступ к памяти.

Второй шаг можно реализовать, используя простой цикл. Но давайте реализуем его используя развернутый цикл:

```

#include <stdint.h>
#include <stdio.h>

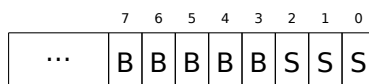
void bzero(uint8_t* dst, size_t count)
{
    int i;

    if (count & (~7))
        // обработать 8-байтные блоки
        for (i=0; i<count>>3; i++)
        {
            *(uint64_t*)dst=0;
            dst=dst+8;
        };

    // обработать хвост
    switch(count & 7)
    {
    case 7: *dst++ = 0;
    case 6: *dst++ = 0;
    case 5: *dst++ = 0;
    case 4: *dst++ = 0;
    case 3: *dst++ = 0;
    case 2: *dst++ = 0;
    case 1: *dst++ = 0;
    case 0: // ничего не делать
            break;
    }
}

```

В начале разберемся, как происходят вычисления. Размер региона в памяти приходит в 64-битном значении. И это значение можно разделить на две части:



( «B» это количество 8-байтных блоков и «S» это длина хвоста в байтах ).

Если разделить размер входного блока в памяти на 8, то значение просто сдвигается на 3 бита вправо. Но для вычисления остатка, нам нужно просто изолировать младшие 3 бита! Так что количество 8-байтных блоков вычисляется как  $count \gg 3$ , а остаток как  $count \& 7$ . В начале, нужно определить, будем ли мы вообще исполнять 8-байтную процедуру, так что нам нужно узнать, не больше ли  $count$  чем 7. Мы делаем это очищая младшие 3 бита и сравнивая результат с нулем, потому что, всё что нам нужно узнать, это ответ на вопрос, содержит ли старшая часть значения  $count$  ненулевые биты. Конечно, это работает потому что 8 это  $2^3$ , так что деление на числа вида  $2^n$  это легко. Это невозможно с другими числами.

А на самом деле, трудно сказать, стоит ли пользоваться такими хакерскими трюками, потому что они приводят к коду, который затем тяжело читать.

С другой стороны, эти трюки очень популярны и практикующий программист,

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

хотя может и не использовать их, всё же должен их понимать.

Так что первая часть простая: получить количество 8-байтных блоков и записать 64-битные нулевые значения в память.

Вторая часть — это развернутый цикл реализованный как fallthrough-выражение switch().

В начале, выразим на понятном русском языке, что мы хотим сделать.

Мы должны «записать столько нулевых байт в память, сколько указано в значении *count*&7».

Если это 0, перейти на конец, больше ничего делать не нужно.

Если это 1, перейти на место внутри выражения switch(), где произойдет только одна операция записи.

Если это 2, перейти на другое место, где две операции записи будут исполнены, и т. д. 7 во входном значении приведет к тому что исполнятся все 7 операций.

8 здесь нет, потому что регион памяти размером в 8 байт будет обработан первой частью нашей функции.

Так что мы сделали развернутый цикл. Это однозначно работало быстрее обычных циклов на старых компьютерах (и наоборот, на современных процессорах короткие циклы работают быстрее развернутых).

Может быть, это всё еще может иметь смысл на современных маломощных дешевых **MCU**<sup>9</sup>.

Посмотрим, что сделает оптимизирующий MSVC 2012:

```
dst$ = 8
count$ = 16
bzero PROC
    test    rdx, -8
    je     SHORT $LN11@bzero
; обработать 8-байтные блоки
    xor    r10d, r10d
    mov    r9, rdx
    shr    r9, 3
    mov    r8d, r10d
    test   r9, r9
    je     SHORT $LN11@bzero
    npad   5
$LL19@bzero:
    inc    r8d
    mov    QWORD PTR [rcx], r10
    add    rcx, 8
    movsxd rax, r8d
    cmp    rax, r9
    jnb   SHORT $LL19@bzero
$LN11@bzero:
```

<sup>9</sup>**MCU!**

```

; обработать хвост
    and     edx, 7
    dec     rdx
    cmp     rdx, 6
    ja     SHORT $LN9@bzero
    lea    r8, OFFSET FLAT:__ImageBase
    mov    eax, DWORD PTR $LN22@bzero[r8+rdx*4]
    add    rax, r8
    jmp    rax
$LN8@bzero:
    mov    BYTE PTR [rcx], 0
    inc    rcx
$LN7@bzero:
    mov    BYTE PTR [rcx], 0
    inc    rcx
$LN6@bzero:
    mov    BYTE PTR [rcx], 0
    inc    rcx
$LN5@bzero:
    mov    BYTE PTR [rcx], 0
    inc    rcx
$LN4@bzero:
    mov    BYTE PTR [rcx], 0
    inc    rcx
$LN3@bzero:
    mov    BYTE PTR [rcx], 0
    inc    rcx
$LN2@bzero:
    mov    BYTE PTR [rcx], 0
$LN9@bzero:
    fatret 0
    npad   1
$LN22@bzero:
    DD     $LN2@bzero
    DD     $LN3@bzero
    DD     $LN4@bzero
    DD     $LN5@bzero
    DD     $LN6@bzero
    DD     $LN7@bzero
    DD     $LN8@bzero
bzero   ENDP

```

Первая часть функции выглядит для нас предсказуемо.

Вторая часть — это просто развернутый цикл и переход передает управление на нужную инструкцию внутри него.

Между парами инструкций MOV/INC никакого другого кода нет, так что исполнение продолжается до самого конца, исполняются столько пар, сколько нужно.

Кстати, мы можем заметить, что пара MOV/INC занимает какое-то фиксированное количество байт (3+3).

Так что пара занимает 6 байт. Зная это, мы можем избавиться от таблицы пере-

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



ходов в `switch()`, мы можем просто умножить входное значение на 6 и перейти на текущий `_RIP + входное_значение * 6`.

Это будет также быстрее, потому что не нужно будет загружать элемент из таблицы переходов (*jump table*).

Может быть, 6 не самая подходящая константа для быстрого умножения, и может быть оно того и не стоит, но вы поняли идею<sup>10</sup>.

Так в прошлом делали с развернутыми циклами олд-скульные демомейкеры.

### 3.9.1. Нужно ли использовать развернутые циклы?

Развернутые циклы могут иметь преимущества если между **RAM!** и **CPU!** нет быстрой кэш-памяти и **CPU!**, чтобы прочитать код очередной инструкции, должен загружать её каждый раз из **RAM!**. Это случай современных маломощных **MCU!** и старых **CPU!**.

Развернутые циклы будут работать медленнее коротких циклов, если есть быстрый кэш между **RAM!** и **CPU!** и тело цикла может поместиться в кэш и CPU будет загружать код оттуда не трогая **RAM!**. Быстрые циклы это циклы у которых тело помещается в L1-кэш, но еще более быстрые циклы это достаточно маленькие, чтобы поместиться в кэш микроопераций.

## 3.10. Деление используя умножение

Простая функция:

```
int f(int a)
{
    return a/9;
};
```

### 3.10.1. x86

...компилируется вполне предсказуемо:

Листинг 3.19: MSVC

```
_a$ = 8          ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    cdq                    ; знаковое расширение EAX до EDX:EAX
    mov     ecx, 9
    idiv   ecx
    pop     ebp
```

<sup>10</sup>В качестве упражнения, вы можете попробовать переработать этот код и избавиться от таблицы переходов. Пару инструкций тоже можно переписать так что они будут занимать 4 байта или 8. 1 байт тоже возможен (используя инструкцию `STOSB`).

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
ret 0
_f ENDP
```

IDIV делит 64-битное число хранящееся в паре регистров EDX:EAX на значение в ECX. В результате, EAX будет содержать **частное**, а EDX— остаток от деления. Результат возвращается из функции через EAX, так что после операции деления, это значение не перекладывается больше никуда, оно уже там, где надо.

Из-за того, что IDIV требует пару регистров EDX:EAX, то перед этим инструкция CDQ расширяет EAX до 64-битного значения учитывая знак, так же, как это делает MOVSX.

Со включенной оптимизацией (/Ox) получается:

Листинг 3.20: Оптимизирующий MSVC

```
_a$ = 8 ; size = 4
_f PROC

    mov     ecx, DWORD PTR _a$[esp-4]
    mov     eax, 954437177 ; 38e38e39H
    imul   ecx
    sar    edx, 1
    mov     eax, edx
    shr    eax, 31 ; 0000001fH
    add    eax, edx
    ret    0
_f ENDP
```

Это — деление через умножение. Умножение конечно быстрее работает. Поэтому можно используя этот трюк <sup>11</sup> создать код эквивалентный тому что мы хотим и работающий быстрее.

В оптимизации компиляторов, это также называется «strength reduction».

GCC 4.4.1 даже без включенной оптимизации генерирует примерно такой же код, как и MSVC с оптимизацией:

Листинг 3.21: Неоптимизирующий GCC 4.4.1

```
public f
f proc near

arg_0 = dword ptr 8

    push   ebp
    mov    ebp, esp
    mov    ecx, [ebp+arg_0]
    mov    edx, 954437177 ; 38E38E39h
    mov    eax, ecx
    imul  edx
    sar   edx, 1
    mov   eax, ecx
```

<sup>11</sup>Читайте подробнее о делении через умножение в [Henry S. Warren, *Hacker's Delight*, (2002)10-3]

```

sar    eax, 1Fh
mov    ecx, edx
sub    ecx, eax
mov    eax, ecx
pop    ebp
retn
f      endp

```

### 3.10.2. Как это работает

Из школьной математики, мы можем вспомнить, что деление на 9 может быть заменено на умножение на  $\frac{1}{9}$ . На самом деле, для чисел с плавающей точкой, иногда компиляторы так и делают, например, инструкция FDIV в x86-коде может быть заменена на FMUL. По крайней мере MSVC 6.0 заменяет деление на 9 на умножение на 0.111111... и иногда нельзя быть уверенным в том, какая операция была в оригинальном исходном коде.

Но когда мы работаем с целочисленными значениями и целочисленными регистрами CPU, мы не можем использовать дроби. Но мы можем переписать дробь так:

$$result = \frac{x}{9} = x \cdot \frac{1}{9} = x \cdot \frac{1 \cdot MagicNumber}{9 \cdot MagicNumber}$$

Учитывая тот факт, что деление на  $2^n$  очень быстро (при помощи сдвигов), теперь нам нужно найти такой *MagicNumber*, для которого следующее уравнение будет справедливо:  $2^n = 9 \cdot MagicNumber$ .

Деление на  $2^{32}$  в каком-то смысле скрыто: младшие 32 бита произведения в EAX не используются (выкидываются), только старшие 32 бита произведения (в EDX) используются и затем сдвигаются еще на 1 бит.

Другими словами, только что увиденный код на ассемблере умножает на  $\frac{954437177}{2^{32+1}}$ , или делит на  $\frac{2^{32+1}}{954437177}$ . Чтобы найти делитель, нужно просто разделить числитель на знаменатель. Используя Wolfram Alpha, мы получаем результат 8.99999999... (что близко к 9).

Читайте больше об этом в [Henry S. Warren, *Hacker's Delight*, (2002)10-3].

Многие люди не замечают "скрытое" деление на  $2^{32}$  или  $2^{64}$ , когда младшая 32-битная часть произведения (или 64-битная) не используется. Поэтому деление через умножение в коде поначалу трудно для понимания.

В Mathematical Recipes<sup>12</sup> есть еще одно объяснение.

### 3.10.3. ARM

В процессоре ARM, как и во многих других «чистых» (pure) RISC-процессорах нет инструкции деления. Нет также возможности умножения на 32-битную константу одной инструкцией (вспомните что 32-битная константа просто не поместится в 32-битных опкод).

<sup>12</sup><https://math.recipes>

При помощи этого любопытного трюка (или хака)<sup>13</sup>, можно обойтись только тремя действиями: сложением, вычитанием и битовыми сдвигами (?? (стр. ??)).

Пример деления 32-битного числа на 10 из [Advanced RISC Machines Ltd, *The ARM Cookbook*, (1994)3.3 Division by a Constant]. На выходе и частное и остаток.

```

; takes argument in a1
; returns quotient in a1, remainder in a2
; cycles could be saved if only divide or remainder is required
SUB    a2, a1, #10           ; keep (x-10) for later
SUB    a1, a1, a1, lsr #2
ADD    a1, a1, a1, lsr #4
ADD    a1, a1, a1, lsr #8
ADD    a1, a1, a1, lsr #16
MOV    a1, a1, lsr #3
ADD    a3, a1, a1, asl #2
SUBS   a2, a2, a3, asl #1    ; calc (x-10) - (x/10)*10
ADDPL  a1, a1, #1           ; fix-up quotient
ADDMI  a2, a2, #10         ; fix-up remainder
MOV    pc, lr

```

### Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

```

__text:00002C58 39 1E 08 E3 E3 18 43 E3  MOV    R1, 0x38E38E39
__text:00002C60 10 F1 50 E7                SMMUL  R0, R0, R1
__text:00002C64 C0 10 A0 E1                MOV    R1, R0,ASR#1
__text:00002C68 A0 0F 81 E0                ADD    R0, R1, R0,LSR#31
__text:00002C6C 1E FF 2F E1                BX     LR

```

Этот код почти тот же, что сгенерирован MSVC и GCC в режиме оптимизации.

Должно быть, LLVM использует тот же алгоритм для поиска констант.

Наблюдательный читатель может спросить, как MOV записала в регистр сразу 32-битное число, ведь это невозможно в режиме ARM.

Действительно невозможно, но как мы видим, здесь на инструкцию 8 байт вместо стандартных 4-х, на самом деле, здесь 2 инструкции.

Первая инструкция загружает в младшие 16 бит регистра значение 0x8E39, а вторая инструкция, на самом деле MOVТ, загружающая в старшие 16 бит регистра значение 0x383E.

**IDA!** легко распознала эту последовательность и для краткости, сократила всё это до одной «псевдо-инструкции».

Инструкция SMMUL (*Signed Most Significant Word Multiply*) умножает числа считая их знаковыми (signed) и оставляет в R0 старшие 32 бита результата, не сохраняя младшие 32 бита.

Инструкция «MOV R1, R0,ASR#1» это арифметический сдвиг право на один бит.

«ADD R0, R1, R0,LSR#31» это  $R0 = R1 + R0 \gg 31$

<sup>13</sup>hack

Дело в том, что в режиме ARM нет отдельных инструкций для битовых сдвигов. Вместо этого, некоторые инструкции (MOV, ADD, SUB, RSB)<sup>14</sup> могут быть дополнены суффиксом, сдвигать ли второй операнд и если да, то на сколько и как. ASR означает *Arithmetic Shift Right*, LSR — *Logical Shift Right*.

### Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

MOV	R1, 0x38E38E39
SMMUL.W	R0, R0, R1
ASRS	R1, R0, #1
ADD.W	R0, R1, R0, LSR#31
BX	LR

В режиме Thumb отдельные инструкции для битовых сдвигов есть, и здесь применяется одна из них — ASRS (арифметический сдвиг вправо).

### Неоптимизирующий Xcode 4.6.3 (LLVM) и Keil 6/2013

Неоптимизирующий LLVM не занимается генерацией подобного кода, а вместо этого просто вставляет вызов библиотечной функции `__divsi3`.

А Keil во всех случаях вставляет вызов функции `__aeabi_idivmod`.

### 3.10.4. MIPS

По какой-то причине, оптимизирующий GCC 4.4.5 сгенерировал просто инструкцию деления:

Листинг 3.22: Оптимизирующий GCC 4.4.5 (IDA)

```
f:
    li    $v0, 9
    bnez  $v0, loc_10
    div   $a0, $v0 ; branch delay slot
    break 0x1C00 ; "break 7" в ассемблерном выводе и в
objdump

loc_10:
    mflo  $v0
    jr    $ra
    or    $at, $zero ; branch delay slot, NOP
```

И кстати, мы видим новую инструкцию: BREAK. Она просто генерирует исключение.

В этом случае, исключение генерируется если делитель 0 (потому что в обычной математике нельзя делить на ноль).

Но компилятор GCC наверное не очень хорошо оптимизировал код, и не заметил, что \$V0 не бывает нулем. Так что проверка осталась здесь.

<sup>14</sup>Эти инструкции также называются «data processing instructions»

Так что если \$V0 будет каким-то образом 0, будет исполнена BREAK, сигнализирующая в **OS!** об исключении.

В противном случае, выполняется MFLO, берущая результат деления из регистра LO и копирующая его в \$V0.

Кстати, как мы уже можем знать, инструкция MUL оставляет старшую 32-битную часть результата в регистре HI и младшую 32-битную часть в LO.

DIV оставляет результат в регистре LO и остаток в HI.

Если изменить выражение на «a % 9», вместо инструкции MFLO будет использована MFHI.

### 3.10.5. Упражнение

- <http://challenges.re/27>

## 3.11. Конверсия строки в число (atoi())

Попробуем реализовать стандарту функцию Си atoi().

### 3.11.1. Простой пример

Это самый простой способ прочитать число, представленное в кодировке **ASCII!**

Он не защищен от ошибок: символ отличный от цифры приведет к неверному результату.

```
#include <stdio.h>

int my_atoi (char *s)
{
    int rt=0;

    while (*s)
    {
        rt=rt*10 + (*s-'0');
        s++;
    };

    return rt;
};

int main()
{
    printf ("%d\n", my_atoi ("1234"));
    printf ("%d\n", my_atoi ("1234567890"));
};
```

То, что делает алгоритм это просто считывает цифры слева направо.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Символ нуля в **ASCII!** вычитается из каждой цифры.

Цифры от «0» до «9» расположены по порядку в таблице **ASCII!**, так что мы даже можем и не знать точного значения символа «0».

Всё что нам нужно знать это то что «0» минус «0» — это 0, а «9» минус «0» это 9, и т. д.

Вычитание «0» от каждого символа в итоге дает число от 0 до 9 включительно.

Любой другой символ, конечно, приведет к неверному результату!

Каждая цифра добавляется к итоговому результату (в переменной «rt»), но итоговый результат также умножается на 10 на каждой цифре.

Другими словами, на каждой итерации, результат сдвигается влево на одну позицию в десятичном виде.

Самая последняя цифра прибавляется, но не сдвигается.

### Оптимизирующий MSVC 2013 x64

Листинг 3.23: Оптимизирующий MSVC 2013 x64

```
s$ = 8
my_atoi PROC
; загрузить первый символ
    movzx    r8d, BYTE PTR [rcx]
; EAX выделен для переменной "rt"
; в начале там 0
    xor     eax, eax
; первый символ - это нулевой байт, т.е., конец строки?
; тогда выходим.
    test    r8b, r8b
    je     SHORT $LN9@my_atoi
$LL2@my_atoi:
    lea    edx, DWORD PTR [rax+rax*4]
; EDX=RAX+RAX*4=rt+rt*4=rt*5
    movsx  eax, r8b
; EAX=входной символ
; загрузить следующий символ в R8D
    movzx  r8d, BYTE PTR [rcx+1]
; передвинуть указатель в RCX на следующий символ:
    lea    rcx, QWORD PTR [rcx+1]
    lea    eax, DWORD PTR [rax+rdx*2]
; EAX=RAX+RDX*2=входной символ + rt*5*2=входной символ + rt*10
; скорректировать цифру вычитая 48 (0x30 или '0')
    add    eax, -48 ; ffffffffffffffff0H
; последний символ был нулем?
    test   r8b, r8b
; перейти на начало цикла, если нет
    jne   SHORT $LL2@my_atoi
$LN9@my_atoi:
    ret    0
my_atoi ENDP
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Символы загружаются в двух местах: первый символ и все последующие символы.

Это сделано для перегруппировки цикла. Здесь нет инструкции для умножения на 10, вместо этого две LEA делают это же.

MSVC иногда использует инструкцию ADD с отрицательной константой вместо SUB.

Это тот случай. Честно говоря, трудно сказать, чем это лучше, чем SUB.

Но MSVC делает так часто.

### Оптимизирующий GCC 4.9.1 x64

Оптимизирующий GCC 4.9.1 более краток, но здесь есть одна лишняя инструкция RET в конце.

Одной было бы достаточно.

Листинг 3.24: Оптимизирующий GCC 4.9.1 x64

```
my_atoi:
; загрузить входной символ в EDX
    movsx    edx, BYTE PTR [rdi]
; EAX выделен для переменной "rt"
    xor     eax, eax
; выйти, если загруженный символ - это нулевой байт
    test    dl, dl
    je     .L4
.L3:
    lea    eax, [rax+rax*4]
; EAX=RAX*5=rt*5
; передвинуть указатель на следующий символ:
    add    rdi, 1
    lea    eax, [rdx-48+rax*2]
; EAX=входной символ - 48 + RAX*2 = входной символ - '0' + rt*10
; загрузить следующий символ:
    movsx    edx, BYTE PTR [rdi]
; перейти на начало цикла, если загруженный символ - это не нулевой байт
    test    dl, dl
    jne    .L3
    rep    ret
.L4:
    rep    ret
```

### Оптимизирующий Keil 6/2013 (Режим ARM)

Листинг 3.25: Оптимизирующий Keil 6/2013 (Режим ARM)

```
my_atoi PROC
; R1 будет содержать указатель на символ
    MOV     r1,r0
; R0 будет содержать переменную "rt"
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

MOV    r0,#0
B      |L0.28|
|L0.12|
ADD    r0,r0,r0,LSL #2
; R0=R0+R0<<2=rt*5
ADD    r0,r2,r0,LSL #1
; R0=входной символ + rt*5<<1 = входной символ + rt*10
; скорректировать, вычитая '0' из rt:
SUB    r0,r0,#0x30
; сдвинуть указатель на следующий символ:
ADD    r1,r1,#1
|L0.28|
; загрузить входной символ в R2
LDRB   r2,[r1,#0]
; это нулевой байт? если нет, перейти на начало цикла.
CMP    r2,#0
BNE    |L0.12|
; выйти, если это нулевой байт.
; переменная "rt" всё еще в регистре R0, готовая для использования в
; вызывающей ф-ции
BX     lr
ENDP

```

### Оптимизирующий Keil 6/2013 (Режим Thumb)

Листинг 3.26: Оптимизирующий Keil 6/2013 (Режим Thumb)

```

my_atoi PROC
; R1 будет указателем на входной символ
MOVVS  r1,r0
; R0 выделен для переменной "rt"
MOVVS  r0,#0
B      |L0.16|
|L0.6|
MOVVS  r3,#0xa
; R3=10
MULS   r0,r3,r0
; R0=R3*R0=rt*10
; передвинуть указатель на следующий символ:
ADDS   r1,r1,#1
; скорректировать, вычитая символ нуля:
SUBS   r0,r0,#0x30
ADDS   r0,r2,r0
; rt=R2+R0=входной символ + (rt*10 - '0')
|L0.16|
; загрузить входной символ в R2
LDRB   r2,[r1,#0]
; это ноль?
CMP    r2,#0
; перейти на тело цикла, если нет
BNE    |L0.6|
; переменная rt сейчас в R0, готовая для использования в вызывающей ф-ции
BX     lr

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

ENDP

Интересно, из школьного курса математики мы можем помнить, что порядок операций сложения и вычитания не играет роли.

Это наш случай: в начале вычисляется выражение  $rt * 10 - '0'$ , затем к нему прибавляется значение входного символа.

Действительно, результат тот же, но компилятор немного всё перегруппировал.

### Оптимизирующий GCC 4.9.1 ARM64

Компилятор для ARM64 может использовать суффикс инструкции, задающий пре-инкремент:

Листинг 3.27: Оптимизирующий GCC 4.9.1 ARM64

```

my_atoi:
; загрузить входной символ в W1
    ldrb    w1, [x0]
    mov     x2, x0
; X2=адрес входной строки
; загруженный символ - 0?
; перейти на выход, если это так
; W1 будет содержать 0 в этом случае.
; он будет перезагружен в W0 на L4.
    cbz    w1, .L4
; W0 будет содержать переменную "rt"
; инициализировать её нулем:
    mov    w0, 0
.L3:
; вычсть 48 или '0' из входной переменной и оставить результат в W3:
    sub    w3, w1, #48
; загрузить следующий символ по адресу X2+1 в W1
; с пре-инкрементом:
    ldrb    w1, [x2,1]!
    add    w0, w0, w0, lsl 2
; W0=W0+W0<<2=W0+W0*4=rt*5
    add    w0, w3, w0, lsl 1
; W0=входная цифра + W0<<1 = входная цифра + rt*5*2 = входная цифра + rt*10
; если только что загруженный символ - это не нулевой байт, перейти на начало
цикла
    cbnz   w1, .L3
; значение для возврата (rt) в W0, готовое для использования в вызывающей
ф-ции
    ret
.L4:
    mov    w0, w1
    ret

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

### 3.11.2. Немного расширенный пример

Новый пример более расширенный, теперь здесь есть проверка знака «минус» в самом начале, и еще он может сообщать об ошибке если не-цифра была найдена во входной строке:

```
#include <stdio.h>

int my_atoi (char *s)
{
    int negative=0;
    int rt=0;

    if (*s=='-')
    {
        negative=1;
        s++;
    };

    while (*s)
    {
        if (*s<'0' || *s>'9')
        {
            printf ("Error! Unexpected char: '%c'\n", *s);
            exit(0);
        };
        rt=rt*10 + (*s-'0');
        s++;
    };

    if (negative)
        return -rt;
    return rt;
};

int main()
{
    printf ("%d\n", my_atoi ("1234"));
    printf ("%d\n", my_atoi ("1234567890"));
    printf ("%d\n", my_atoi ("-1234"));
    printf ("%d\n", my_atoi ("-1234567890"));
    printf ("%d\n", my_atoi ("-a1234567890")); // error
};
```

### Оптимизирующий GCC 4.9.1 x64

Листинг 3.28: Оптимизирующий GCC 4.9.1 x64

```
.LC0:
    .string "Error! Unexpected char: '%c'\n"

my_atoi:
    sub    rsp, 8
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    movsx    edx, BYTE PTR [rdi]
; проверка на знак минуса
    cmp     dl, 45 ; '-'
    je     .L22
    xor     esi, esi
    test    dl, dl
    je     .L20
.L10:
; ESI=0 здесь, если знака минуса не было, или 1 в противном случае
    lea    eax, [rdx-48]
; любой символ, отличающийся от цифры в результате
; даст беззнаковое число больше 9 после вычитания
; так что если это не число, перейти на L4,
; где будет просигнализировано об ошибке:
    cmp     al, 9
    ja     .L4
    xor     eax, eax
    jmp    .L6
.L7:
    lea    ecx, [rdx-48]
    cmp    cl, 9
    ja     .L4
.L6:
    lea    eax, [rax+rax*4]
    add    rdi, 1
    lea    eax, [rdx-48+rax*2]
    movsx  edx, BYTE PTR [rdi]
    test   dl, dl
    jne    .L7
; если знака минуса не было, пропустить инструкцию NEG
; а если был, то исполнить её.
    test   esi, esi
    je     .L18
    neg    eax
.L18:
    add    rsp, 8
    ret
.L22:
    movsx  edx, BYTE PTR [rdi+1]
    lea    rax, [rdi+1]
    test   dl, dl
    je     .L20
    mov    rdi, rax
    mov    esi, 1
    jmp    .L10
.L20:
    xor    eax, eax
    jmp    .L18
.L4:
; сообщить об ошибке. символ в EDX
    mov    edi, 1
    mov    esi, OFFSET FLAT:.LC0 ; "Error! Unexpected char: '%c'\n"
    xor    eax, eax

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

call    __printf_chk
xor     edi, edi
call    exit

```

Если знак «минус» был найден в начале строки, инструкция NEG будет исполнена в конце.

Она просто меняет знак числа.

Еще кое-что надо отметить. Как среднестатистический программист будет проверять, является ли символ цифрой?

Так же, как и у нас в исходном коде:

```

if (*s<'0' || *s>'9')
    ...

```

Здесь две операции сравнения. Но что интересно, так это то что мы можем заменить обе операции на одну:

просто вычитайте «0» из значения символа, считается результат за беззнаковое значение (это важно) и проверьте, не больше ли он чем 9.

Например, скажем, строка на входе имеет символ точки («.»), которая имеет код 46 в таблице **ASCII!**

$46 - 48 = -2$  если считать результат за знаковое число. Действительно, символ точки расположен на два места раньше, чем символ «0» в таблице **ASCII!**

Но это  $0xFFFFFFFF$  (4294967294) если считать результат за беззнаковое значение, и это точно больше чем 9!

Компиляторы часто так делают, важно распознавать эти трюки.

Еще один пример подобного в этой книге: ?? (стр. ??).

Оптимизирующий MSVC 2013 x64 применяет те же трюки.

### Оптимизирующий Keil 6/2013 (Режим ARM)

Листинг 3.29: Оптимизирующий Keil 6/2013 (Режим ARM)

```

1 my_atoi PROC
2     PUSH    {r4-r6,lr}
3     MOV     r4,r0
4     LDRB   r0,[r0,#0]
5     MOV     r6,#0
6     MOV     r5,r6
7     CMP    r0,#0x2d '-'
8 ; R6 будет содержать 1 если минус был встречен, или 0 в противном случае
9     MOVEQ  r6,#1
10    ADDEQ  r4,r4,#1
11    B      |L0.80|
12 |L0.36|
13    SUB    r0,r1,#0x30
14    CMP    r0,#0xa

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

15      BCC      |L0.64|
16      ADR      r0, |L0.220|
17      BL       __2printf
18      MOV      r0, #0
19      BL       exit
20 |L0.64|
21      LDRB     r0, [r4], #1
22      ADD      r1, r5, r5, LSL #2
23      ADD      r0, r0, r1, LSL #1
24      SUB      r5, r0, #0x30
25 |L0.80|
26      LDRB     r1, [r4, #0]
27      CMP      r1, #0
28      BNE      |L0.36|
29      CMP      r6, #0
30 ; поменять знак в переменной результата
31      RSBNE    r0, r5, #0
32      MOVEQ    r0, r5
33      POP      {r4-r6, pc}
34      ENDP
35
36 |L0.220|
37      DCB      "Error! Unexpected char: '%c'\n", 0

```

В 32-битном ARM нет инструкции NEG, так что вместо этого используется операция «Reverse Subtraction» (строка 31).

Она сработает если результат инструкции CMP (на строке 29) был «Not Equal» (не равно, отсюда суффикс -NE suffix).

Что делает RSBNE это просто вычитает результирующее значение из нуля.

Она работает, как и обычное вычитание, но меняет местами операнды.

Вычитание любого числа из нуля это смена знака:  $0 - x = -x$ .

Код для режима Thumb почти такой же.

GCC 4.9 для ARM64 может использовать инструкцию NEG, доступную в ARM64.

### 3.11.3. Упражнение

Кстати, security research-еры часто имеют дело с непредсказуемым поведением программ во время обработки некорректных данных. Например, во время fuzzing-а.

В качестве упражнения, вы можете попробовать ввести символы не относящиеся к числам и посмотреть, что случится.

Попробуйте объяснить, что произошло, и почему.

## 3.12. Inline-функции

Inline-код это когда компилятор, вместо того чтобы генерировать инструкцию вызова небольшой функции, просто вставляет её тело прямо в это место.

Листинг 3.30: Простой пример

```
#include <stdio.h>

int celsius_to_fahrenheit (int celsius)
{
    return celsius * 9 / 5 + 32;
};

int main(int argc, char *argv[])
{
    int celsius=atol(argv[1]);
    printf ("%d\n", celsius_to_fahrenheit (celsius));
};
```

...это компилируется вполне предсказуемо, хотя, если включить оптимизации GCC (-O3), мы увидим:

Листинг 3.31: Оптимизирующий GCC 4.8.1

```
_main:
    push    ebp
    mov     ebp, esp
    and     esp, -16
    sub     esp, 16
    call   __main
    mov     eax, DWORD PTR [ebp+12]
    mov     eax, DWORD PTR [eax+4]
    mov     DWORD PTR [esp], eax
    call   _atol
    mov     edx, 1717986919
    mov     DWORD PTR [esp], OFFSET FLAT:LC2 ; "%d\n"
    lea    ecx, [eax+eax*8]
    mov     eax, ecx
    imul   edx
    sar    ecx, 31
    sar    edx
    sub    edx, ecx
    add    edx, 32
    mov    DWORD PTR [esp+4], edx
    call  _printf
    leave
    ret
```

(Здесь деление заменено умножением(?? (стр. ??)).)

Да, наша маленькая функция `celsius_to_fahrenheit()` была помещена прямо перед вызовом `printf()`.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Почему? Это может быть быстрее чем исполнять код самой функции плюс затраты на вызов и возврат.

Современные оптимизирующие компиляторы самостоятельно выбирают функции для вставки. Но компилятор можно дополнительно принудить развернуть некоторую функцию, если маркировать её ключевым словом «inline» в её определении.

### 3.12.1. Функции работы со строками и памятью

Другая очень частая оптимизация это вставка кода строковых функций таких как *strcpy()*, *strcmp()*, *strlen()*, *memset()*, *memcpy()*, *memcpu()*, итд.

Иногда это быстрее, чем вызывать отдельную функцию.

Это очень часто встречающиеся шаблонные вставки, которые желательно распознавать reverse engineer-ам «на глаз».

#### strcmp()

Листинг 3.32: пример с strcmp()

```
bool is_bool (char *s)
{
    if (strcmp (s, "true")==0)
        return true;
    if (strcmp (s, "false")==0)
        return false;

    assert(0);
};
```

Листинг 3.33: Оптимизирующий GCC 4.8.1

```
.LC0:
    .string "true"
.LC1:
    .string "false"
is_bool:
.LFB0:
    push    edi
    mov     ecx, 5
    push   esi
    mov     edi, OFFSET FLAT:.LC0
    sub    esp, 20
    mov     esi, DWORD PTR [esp+32]
    repz   cmpsb
    je     .L3
    mov     esi, DWORD PTR [esp+32]
    mov     ecx, 6
    mov     edi, OFFSET FLAT:.LC1
    repz   cmpsb
    seta   cl
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

        setb    dl
        xor     eax, eax
        cmp    cl, dl
        jne    .L8
        add    esp, 20
        pop    esi
        pop    edi
        ret

.L8:
        mov    DWORD PTR [esp], 0
        call  assert
        add    esp, 20
        pop    esi
        pop    edi
        ret

.L3:
        add    esp, 20
        mov    eax, 1
        pop    esi
        pop    edi
        ret

```

Листинг 3.34: Оптимизирующий MSVC 2010

```

$SG3454 DB 'true', 00H
$SG3456 DB 'false', 00H

_s$ = 8 ; size = 4
?is_bool@@YA_NPAD@Z PROC ; is_bool
        push   esi
        mov    esi, DWORD PTR _s$[esp]
        mov    ecx, OFFSET $SG3454 ; 'true'
        mov    eax, esi
        npad  4 ; выровнять следующую метку
$LL6@is_bool:
        mov    dl, BYTE PTR [eax]
        cmp    dl, BYTE PTR [ecx]
        jne    $LN7@is_bool
        test   dl, dl
        je     $LN8@is_bool
        mov    dl, BYTE PTR [eax+1]
        cmp    dl, BYTE PTR [ecx+1]
        jne    $LN7@is_bool
        add    eax, 2
        add    ecx, 2
        test   dl, dl
        jne    $LL6@is_bool
$LN8@is_bool:
        xor    eax, eax
        jmp    $LN9@is_bool
$LN7@is_bool:
        sbb   eax, eax
        sbb   eax, -1
$LN9@is_bool:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    test    eax, eax
    jne     SHORT $LN2@is_bool

    mov     al, 1
    pop     esi

    ret     0
$LN2@is_bool:

    mov     ecx, OFFSET $SG3456 ; 'false'
    mov     eax, esi
$LL10@is_bool:
    mov     dl, BYTE PTR [eax]
    cmp     dl, BYTE PTR [ecx]
    jne     SHORT $LN11@is_bool
    test    dl, dl
    je      SHORT $LN12@is_bool
    mov     dl, BYTE PTR [eax+1]
    cmp     dl, BYTE PTR [ecx+1]
    jne     SHORT $LN11@is_bool
    add     eax, 2
    add     ecx, 2
    test    dl, dl
    jne     SHORT $LL10@is_bool
$LN12@is_bool:
    xor     eax, eax
    jmp     SHORT $LN13@is_bool
$LN11@is_bool:
    sbb     eax, eax
    sbb     eax, -1
$LN13@is_bool:
    test    eax, eax
    jne     SHORT $LN1@is_bool

    xor     al, al
    pop     esi

    ret     0
$LN1@is_bool:

    push    11
    push    OFFSET $SG3458
    push    OFFSET $SG3459
    call    DWORD PTR __imp__wassert
    add     esp, 12
    pop     esi

    ret     0
?is_bool@@YA_NPAD@Z ENDP ; is_bool

```

## strlen()

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Листинг 3.35: пример с strlen()

```
int strlen_test(char *s1)
{
    return strlen(s1);
};
```

Листинг 3.36: Оптимизирующий MSVC 2010

```
_s1$ = 8 ; size = 4
_strlen_test PROC
    mov     eax, DWORD PTR _s1$[esp-4]
    lea    edx, DWORD PTR [eax+1]
$LL3@strlen_tes:
    mov     cl, BYTE PTR [eax]
    inc    eax
    test   cl, cl
    jne    SHORT $LL3@strlen_tes
    sub    eax, edx
    ret    0
_strlen_test ENDP
```

**strcpy()**

Листинг 3.37: пример с strcpy()

```
void strcpy_test(char *s1, char *outbuf)
{
    strcpy(outbuf, s1);
};
```

Листинг 3.38: Оптимизирующий MSVC 2010

```
_s1$ = 8 ; size = 4
_outbuf$ = 12 ; size = 4
_strcpy_test PROC
    mov     eax, DWORD PTR _s1$[esp-4]
    mov     edx, DWORD PTR _outbuf$[esp-4]
    sub    edx, eax
    npad   6 ; выровнять следующую метку
$LL3@strcpy_tes:
    mov     cl, BYTE PTR [eax]
    mov     BYTE PTR [edx+eax], cl
    inc    eax
    test   cl, cl
    jne    SHORT $LL3@strcpy_tes
    ret    0
_strcpy_test ENDP
```

**memset()****Пример#1**


---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Листинг 3.39: 32 байта

```
#include <stdio.h>

void f(char *out)
{
    memset(out, 0, 32);
};
```

Многие компиляторы не генерируют вызов `memset()` для коротких блоков, а просто вставляют набор MOV-ов:

Листинг 3.40: Оптимизирующий GCC 4.9.1 x64

```
f:
    mov     QWORD PTR [rdi], 0
    mov     QWORD PTR [rdi+8], 0
    mov     QWORD PTR [rdi+16], 0
    mov     QWORD PTR [rdi+24], 0
    ret
```

Кстати, это напоминает развернутые циклы: ?? (стр. ??).

## Пример#2

Листинг 3.41: 67 байт

```
#include <stdio.h>

void f(char *out)
{
    memset(out, 0, 67);
};
```

Когда размер блока не кратен 4 или 8, разные компиляторы могут вести себя по-разному.

Например, MSVC 2012 продолжает вставлять MOV:

Листинг 3.42: Оптимизирующий MSVC 2012 x64

```
out$ = 8
f     PROC
    xor     eax, eax
    mov     QWORD PTR [rcx], rax
    mov     QWORD PTR [rcx+8], rax
    mov     QWORD PTR [rcx+16], rax
    mov     QWORD PTR [rcx+24], rax
    mov     QWORD PTR [rcx+32], rax
    mov     QWORD PTR [rcx+40], rax
    mov     QWORD PTR [rcx+48], rax
    mov     QWORD PTR [rcx+56], rax
    mov     WORD PTR [rcx+64], ax
    mov     BYTE PTR [rcx+66], al
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
f      ret    0
      ENDP
```

...а GCC использует REP STOSQ, полагая, что так будет короче, чем пачка MOV's:

Листинг 3.43: Оптимизирующий GCC 4.9.1 x64

```
f:
      mov    QWORD PTR [rdi], 0
      mov    QWORD PTR [rdi+59], 0
      mov    rcx, rdi
      lea   rdi, [rdi+8]
      xor    eax, eax
      and    rdi, -8
      sub    rcx, rdi
      add    ecx, 67
      shr    ecx, 3
      rep stosq
      ret
```

## memcpy()

### Короткие блоки

Если нужно скопировать немного байт, то, нередко, memcpy() заменяется на несколько инструкций MOV.

Листинг 3.44: пример с memcpy()

```
void memcpy_7(char *inbuf, char *outbuf)
{
    memcpy(outbuf+10, inbuf, 7);
};
```

Листинг 3.45: Оптимизирующий MSVC 2010

```
_inbuf$ = 8      ; size = 4
_outbuf$ = 12    ; size = 4
_memcpy_7 PROC
      mov    ecx, DWORD PTR _inbuf$[esp-4]
      mov    edx, DWORD PTR [ecx]
      mov    eax, DWORD PTR _outbuf$[esp-4]
      mov    DWORD PTR [eax+10], edx
      mov    dx, WORD PTR [ecx+4]
      mov    WORD PTR [eax+14], dx
      mov    cl, BYTE PTR [ecx+6]
      mov    BYTE PTR [eax+16], cl
      ret    0
_memcpy_7 ENDP
```

Листинг 3.46: Оптимизирующий GCC 4.8.1

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

memcpy_7:
    push    ebx
    mov     eax, DWORD PTR [esp+8]
    mov     ecx, DWORD PTR [esp+12]
    mov     ebx, DWORD PTR [eax]
    lea    edx, [ecx+10]
    mov     DWORD PTR [ecx+10], ebx
    movzx  ecx, WORD PTR [eax+4]
    mov     WORD PTR [edx+4], cx
    movzx  eax, BYTE PTR [eax+6]
    mov     BYTE PTR [edx+6], al
    pop    ebx
    ret

```

Обычно это происходит так: в начале копируются 4-байтные блоки, затем 16-битное слово (если нужно), затем последний байт (если нужно).

Точно так же при помощи MOV копируются структуры: ?? (стр. ??).

### Длинные блоки

Здесь компиляторы ведут себя по-разному.

Листинг 3.47: пример с memcpy()

```

void memcpy_128(char *inbuf, char *outbuf)
{
    memcpy(outbuf+10, inbuf, 128);
};

void memcpy_123(char *inbuf, char *outbuf)
{
    memcpy(outbuf+10, inbuf, 123);
};

```

При копировании 128 байт, MSVC может обойтись одной инструкцией MOVSD (ведь 128 кратно 4):

Листинг 3.48: Оптимизирующий MSVC 2010

```

_inbuf$ = 8           ; size = 4
_outbuf$ = 12        ; size = 4
_memcpy_128 PROC
    push    esi
    mov     esi, DWORD PTR _inbuf$[esp]
    push    edi
    mov     edi, DWORD PTR _outbuf$[esp+4]
    add     edi, 10
    mov     ecx, 32
    rep movsd
    pop     edi
    pop     esi
    ret     0

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
_memcpy_128 ENDP
```

При копировании 123-х байт, в начале копируется 30 32-битных слов при помощи MOVSD (это 120 байт), затем копируется 2 байта при помощи MOVSW, затем еще один байт при помощи MOVSB.

Листинг 3.49: Оптимизирующий MSVC 2010

```
_inbuf$ = 8           ; size = 4
_outbuf$ = 12        ; size = 4
_memcpy_123 PROC
    push    esi
    mov     esi, DWORD PTR _inbuf$[esp]
    push    edi
    mov     edi, DWORD PTR _outbuf$[esp+4]
    add     edi, 10
    mov     ecx, 30
    rep movsd
    movsw
    movsb
    pop     edi
    pop     esi
    ret     0
_memcpy_123 ENDP
```

GCC во всех случаях вставляет большую универсальную функцию, работающую для всех размеров блоков:

Листинг 3.50: Оптимизирующий GCC 4.8.1

```
memcpy_123:
.LFB3:
    push    edi
    mov     eax, 123
    push    esi
    mov     edx, DWORD PTR [esp+16]
    mov     esi, DWORD PTR [esp+12]
    lea    edi, [edx+10]
    test   edi, 1
    jne    .L24
    test   edi, 2
    jne    .L25
.L7:
    mov     ecx, eax
    xor     edx, edx
    shr     ecx, 2
    test   al, 2
    rep movsd
    je     .L8
    movzx  edx, WORD PTR [esi]
    mov    WORD PTR [edi], dx
    mov    edx, 2
.L8:
    test   al, 1
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мой адреса](#). Спасибо!

```

        je      .L5
        movzx  eax, BYTE PTR [esi+edx]
        mov    BYTE PTR [edi+edx], al
.L5:
        pop    esi
        pop    edi
        ret
.L24:
        movzx  eax, BYTE PTR [esi]
        lea   edi, [edx+11]
        add   esi, 1
        test  edi, 2
        mov   BYTE PTR [edx+10], al
        mov   eax, 122
        je    .L7
.L25:
        movzx  edx, WORD PTR [esi]
        add   edi, 2
        add   esi, 2
        sub   eax, 2
        mov   WORD PTR [edi-2], dx
        jmp   .L7
.LFE3:

```

Универсальные функции копирования блоков обычно работают по следующей схеме: вычислить, сколько 32-битных слов можно скопировать, затем сделать это при помощи MOVSD, затем скопировать остатки.

Более сложные функции копирования используют **SIMD!** и учитывают выравнивание в памяти.

Как пример функции strlen() использующую SIMD : ?? (стр. ??).

## memcmp()

Листинг 3.51: пример с memcmp()

```

int memcmp_1235(char *buf1, char *buf2)
{
    return memcmp(buf1, buf2, 1235);
};

```

Для блоков разной длины, MSVC 2013 вставляет одну и ту же универсальную функцию:

Листинг 3.52: Оптимизирующий MSVC 2010

```

_buf1$ = 8      ; size = 4
_buf2$ = 12     ; size = 4
_memcmp_1235 PROC
    mov     ecx, DWORD PTR _buf1$[esp-4]
    mov     edx, DWORD PTR _buf2$[esp-4]
    push   esi
    mov     esi, 1231

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

        npad      2
$LL5@memcmp_123:
        mov     eax, DWORD PTR [ecx]
        cmp     eax, DWORD PTR [edx]
        jne     SHORT $LN4@memcmp_123
        add     ecx, 4
        add     edx, 4
        sub     esi, 4
        jae     SHORT $LL5@memcmp_123
$LN4@memcmp_123:
        mov     al, BYTE PTR [ecx]
        cmp     al, BYTE PTR [edx]
        jne     SHORT $LN6@memcmp_123
        mov     al, BYTE PTR [ecx+1]
        cmp     al, BYTE PTR [edx+1]
        jne     SHORT $LN6@memcmp_123
        mov     al, BYTE PTR [ecx+2]
        cmp     al, BYTE PTR [edx+2]
        jne     SHORT $LN6@memcmp_123
        cmp     esi, -1
        je      SHORT $LN3@memcmp_123
        mov     al, BYTE PTR [ecx+3]
        cmp     al, BYTE PTR [edx+3]
        jne     SHORT $LN6@memcmp_123
$LN3@memcmp_123:
        xor     eax, eax
        pop     esi
        ret     0
$LN6@memcmp_123:
        sbb     eax, eax
        or      eax, 1
        pop     esi
        ret     0
_memcmp_1235 ENDP

```

### strcat()

Это ф-ция strcat() в том виде, в котором её сгенерировала MSVC 6.0. Здесь видны 3 части: 1) измерение длины исходной строки (первый scasb); 2) измерение длины целевой строки (второй scasb); 3) копирование исходной строки в конец целевой (пара movsd/movsb).

Листинг 3.53: strcat()

```

        lea     edi, [src]
        or      ecx, 0FFFFFFFh
        repne  scasb
        not     ecx
        sub     edi, ecx
        mov     esi, edi
        mov     edi, [dst]
        mov     edx, ecx
        or      ecx, 0FFFFFFFh

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

repne scasb
mov     ecx, edx
dec     edi
shr     ecx, 2
rep movsd
mov     ecx, edx
and     ecx, 3
rep movsb

```

### Скрипт для IDA

Есть также небольшой скрипт для **IDA!** для поиска и сворачивания таких очень часто попадающихся inline-функций:

[GitHub](#).

## 3.13. C99 restrict

А вот причина, из-за которой программы на Фортран, в некоторых случаях, работают быстрее чем на Си.

```

void f1 (int* x, int* y, int* sum, int* product, int* sum_product, int* ↗
↳ update_me, size_t s)
{
    for (int i=0; i<s; i++)
    {
        sum[i]=x[i]+y[i];
        product[i]=x[i]*y[i];
        update_me[i]=i*123; // some dummy value
        sum_product[i]=sum[i]+product[i];
    };
};

```

Это очень простой пример, в котором есть одна особенность: указатель на массив `update_me` может быть указателем на массив `sum`, `product`, или даже `sum_product`—ведь нет ничего криминального в том чтобы аргументам функции быть такими, верно?

Компилятор знает об этом, поэтому генерирует код, где в теле цикла будет 4 основных стадии:

- вычислить следующий `sum[i]`
- вычислить следующий `product[i]`
- вычислить следующий `update_me[i]`
- вычислить следующий `sum_product[i]`—на этой стадии придется снова загружать из памяти подсчитанные `sum[i]` и `product[i]`

Возможно ли оптимизировать последнюю стадию? Ведь подсчитанные `sum[i]` и `product[i]` не обязательно снова загружать из памяти, ведь мы их только что подсчитали.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Можно, но компилятор не уверен, что на третьей стадии ничего не затерлось!

Это называется «pointer aliasing», ситуация, когда компилятор не может быть уверен, что память на которую указывает какой-то указатель, не изменилась.

*restrict* в стандарте Си C99 [ISO/IEC 9899:TC3 (C C99 standard), (2007) 6.7.3/1] это обещание, данное компилятору программистом, что аргументы функции, отмеченные этим ключевым словом, всегда будут указывать на разные места в памяти и пересекаться не будут.

Если быть более точным, и описывать это формально, *restrict* показывает, что только данный указатель будет использоваться для доступа к этому объекту, больше никакой указатель для этого использоваться не будет.

Можно даже сказать, что к всякому объекту, доступ будет осуществляться только через один единственный указатель, если он отмечен как *restrict*.

Добавим это ключевое слово к каждому аргументу-указателю:

```
void f2 (int* restrict x, int* restrict y, int* restrict sum, int* restrict
    ↪ product, int* restrict sum_product,
    int* restrict update_me, size_t s)
{
    for (int i=0; i<s; i++)
    {
        sum[i]=x[i]+y[i];
        product[i]=x[i]*y[i];
        update_me[i]=i*123; // some dummy value
        sum_product[i]=sum[i]+product[i];
    };
};
```

Посмотрим результаты:

Листинг 3.54: GCC x64: f1()

```
f1:
    push    r15 r14 r13 r12 rbp rdi rsi rbx
    mov     r13, QWORD PTR 120[rsi]
    mov     rbp, QWORD PTR 104[rsi]
    mov     r12, QWORD PTR 112[rsi]
    test    r13, r13
    je     .L1
    add     r13, 1
    xor     ebx, ebx
    mov     edi, 1
    xor     r11d, r11d
    jmp     .L4
.L6:
    mov     r11, rdi
    mov     rdi, rax
.L4:
    lea    rax, 0[0+r11*4]
    lea    r10, [rcx+rax]
    lea    r14, [rdx+rax]
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

lea    rsi, [r8+rax]
add    rax, r9
mov    r15d, DWORD PTR [r10]
add    r15d, DWORD PTR [r14]
mov    DWORD PTR [rsi], r15d      ; сохранить в sum[]
mov    r10d, DWORD PTR [r10]
imul  r10d, DWORD PTR [r14]
mov    DWORD PTR [rax], r10d     ; сохранить в product[]
mov    DWORD PTR [r12+r11*4], ebx ; сохранить в update_me[]
add    ebx, 123
mov    r10d, DWORD PTR [rsi]     ; перезагрузить sum[i]
add    r10d, DWORD PTR [rax]     ; перезагрузить product[i]
lea    rax, 1[rdi]
cmp    rax, r13
mov    DWORD PTR 0[rbp+r11*4], r10d ; сохранить в sum_product[]
jne    .L6
.L1:
pop    rbx rsi rdi rbp r12 r13 r14 r15
ret

```

Листинг 3.55: GCC x64: f2()

```

f2:
push  r13 r12 rbp rdi rsi rbx
mov   r13, QWORD PTR 104[rsi]
mov   rbp, QWORD PTR 88[rsi]
mov   r12, QWORD PTR 96[rsi]
test  r13, r13
je    .L7
add   r13, 1
xor   r10d, r10d
mov   edi, 1
xor   eax, eax
jmp   .L10
.L11:
mov   rax, rdi
mov   rdi, r11
.L10:
mov   esi, DWORD PTR [rcx+rax*4]
mov   r11d, DWORD PTR [rdx+rax*4]
mov   DWORD PTR [r12+rax*4], r10d ; сохранить в update_me[]
add   r10d, 123
lea   ebx, [rsi+r11]
imul  r11d, esi
mov   DWORD PTR [r8+rax*4], ebx   ; сохранить в sum[]
mov   DWORD PTR [r9+rax*4], r11d ; сохранить в product[]
add   r11d, ebx
mov   DWORD PTR 0[rbp+rax*4], r11d ; сохранить в sum_product[]
lea   r11, 1[rdi]
cmp   r11, r13
jne   .L11
.L7:
pop   rbx rsi rdi rbp r12 r13
ret

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Разница между скомпилированной функцией `f1()` и `f2()` такая: в `f1()`, `sum[i]` и `product[i]` загружаются снова посреди тела цикла, а в `f2()` этого нет, используются уже подсчитанные значения, ведь мы «пообещали» компилятору, что никто и ничто не изменит значения в `sum[i]` и `product[i]` во время исполнения тела цикла, поэтому он «уверен», что значения из памяти можно не загружать снова. Очевидно, второй вариант работает быстрее.

Но что будет если указатели в аргументах функций все же будут пересекаться?

Это на совести программиста, а результаты вычислений будут неверными.

Вернемся к Фортрану. Компиляторы с этого ЯП, по умолчанию, все указатели считают таковыми, поэтому, когда не было возможности указать `restrict` в Си, то компилятор с Фортрана в этих случаях мог генерировать более быстрый код.

Насколько это практично? Там, где функция работает с несколькими большими блоками в памяти.

Такого очень много в линейной алгебре, например.

Очень много линейной алгебры используется на суперкомпьютерах/**НРС!**<sup>15</sup>, возможно, поэтому, традиционно, там часто используется Фортран, до сих пор [Eugene Loh, *The Ideal HPC Programming Language*, (2010)]. Ну а когда итераций цикла не очень много, конечно, тогда прирост скорости может и не быть ощутимым.

## 3.14. Функция `abs()` без переходов

Снова вернемся к уже рассмотренному ранее примеру ?? (стр. ??) и спросим себя, возможно ли сделать версию этого кода под x86 без переходов?

```
int my_abs (int i)
{
    if (i<0)
        return -i;
    else
        return i;
};
```

И ответ положительный.

### 3.14.1. Оптимизирующий GCC 4.9.1 x64

Мы можем это увидеть если скомпилируем оптимизирующим GCC 4.9:

<sup>15</sup>НРС!

## Листинг 3.56: Оптимизирующий GCC 4.9 x64

```

my_abs:
    mov     edx, edi
    mov     eax, edi
    sar    edx, 31
; EDX здесь 0xFFFFFFFF если знак входного значения -- минус
; EDX ноль если знак входного значения -- плюс (включая ноль)
; следующие две инструкции имеют эффект только если EDX равен 0xFFFFFFFF
; либо не работают, если EDX -- ноль
    xor     eax, edx
    sub     eax, edx
    ret

```

И вот как он работает:

Арифметически сдвигаем входное значение вправо на 31.

Арифметический сдвиг означает знаковое расширение, так что если **MSB!** это 1, то все 32 бита будут заполнены единицами, либо нулями в противном случае.

Другими словами, инструкция SAR REG, 31 делает 0xFFFFFFFF если знак был отрицательным либо 0 если положительным.

После исполнения SAR, это значение у нас в EDX.

Затем, если значение 0xFFFFFFFF (т.е. знак отрицательный) входное значение инвертируется (потому что XOR REG, 0xFFFFFFFF работает как операция инвертирования всех бит).

Затем, снова, если значение 0xFFFFFFFF (т.е. знак отрицательный), 1 прибавляется к итоговому результату (потому что вычитание  $-1$  из значения это то же что и инкремент).

Инвертирование всех бит и инкремент, это то, как меняется знак у значения в формате two's complement.

Мы можем заметить, что последние две инструкции делают что-то если знак входного значения отрицательный.

В противном случае (если знак положительный) они не делают ничего, оставляя входное значение нетронутым.

Алгоритм разъяснен в [Henry S. Warren, *Hacker's Delight*, (2002)2-4]. Трудно сказать, как именно GCC сгенерировал его, оптимизировал сам или просто нашел подходящий шаблон среди известных?

### 3.14.2. Оптимизирующий GCC 4.9 ARM64

GCC 4.9 для ARM64 делает почти то же, только использует полные 64-битные регистры.

Здесь меньше инструкций, потому что входное значение может быть сдвинуто используя суффикс инструкции («asr») вместо отдельной инструкции.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

## Листинг 3.57: Оптимизирующий GCC 4.9 ARM64

```

my_abs:
; расширить входное 32-битное значение до 64-битного в регистре X0, учитывая
  знак:
    sxtw    x0, w0
    eor     x1, x0, x0, asr 63
; X1=X0^(X0>>63) (арифметический сдвиг)
    sub     x0, x1, x0, asr 63
; X0=X1-(X0>>63)=X0^(X0>>63)-(X0>>63)
; (все сдвиги -- арифметические)
    ret

```

## 3.15. Функции с переменным количеством аргументов

Функции вроде `printf()` и `scanf()` могут иметь переменное количество аргументов (*variadic*).

Как обращаться к аргументам?

### 3.15.1. Вычисление среднего арифметического

Представим, что нам нужно вычислить [среднее арифметическое](#), и по какой-то странной причине, нам нужно задать все числа в аргументах функции.

Но в Си/Си++ функции с переменным кол-вом аргументов невозможно определить кол-во аргументов, так что обозначим значение `-1` как конец списка.

#### Используя макрос `va_arg`

Имеется стандартный заголовочный файл `stdarg.h`, который определяет макросы для работы с такими аргументами.

Их так же используют функции `printf()` и `scanf()`.

```

#include <stdio.h>
#include <stdarg.h>

int arith_mean(int v, ...)
{
    va_list args;
    int sum=v, count=1, i;
    va_start(args, v);

    while(1)
    {
        i=va_arg(args, int);
        if (i==-1) // терминатор
            break;
        sum=sum+i;
    }
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        count++;
    }

    va_end(args);
    return sum/count;
};

int main()
{
    printf ("%d\n", arith_mean (1, 2, 7, 10, 15, -1 /* терминатор */));
};

```

Самый первый аргумент должен трактоваться как обычный аргумент. Остальные аргументы загружаются используя макрос `va_arg`, и затем суммируются.

Так что внутри?

### Соглашение о вызовах *cdecl*

Листинг 3.58: Оптимизирующий MSVC 6.0

```

_v$ = 8
_arith_mean PROC NEAR
    mov     eax, DWORD PTR _v$[esp-4] ; загрузить первый аргумент в sum
    push   esi
    mov     esi, 1                    ; count=1
    lea    edx, DWORD PTR _v$[esp]  ; адрес первого аргумента
$L838:
    mov     ecx, DWORD PTR [edx+4]   ; загрузить следующий аргумент
    add     edx, 4                   ; сдвинуть указатель на следующий
    аргумент
    cmp     ecx, -1                  ; это -1?
    je     SHORT $L856              ; выйти, если это так
    add     eax, ecx                 ; sum = sum + загруженный аргумент
    inc     esi                      ; count++
    jmp    SHORT $L838
$L856:
    ; вычислить результат деления

    cdq
    idiv   esi
    pop    esi
    ret    0
_arith_mean ENDP

$SG851 DB     '%d', 0aH, 00H

_main PROC NEAR
    push   -1
    push   15
    push   10

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

    push    7
    push    2
    push    1
    call    _arith_mean
    push    eax
    push    OFFSET FLAT:$SG851 ; '%d'
    call    _printf
    add     esp, 32
    ret     0
_main    ENDP

```

Аргументы, как мы видим, передаются в `main()` один за одним.

Первый аргумент заталкивается в локальный стек первым.

Терминатор (оконечивающее значение `-1`) заталкивается последним.

Функция `arith_mean()` берет первый аргумент и сохраняет его значение в переменной `sum`.

Затем, она записывает адрес второго аргумента в регистр `EDX`, берет значение оттуда, прибавляет к `sum`, и делает это в бесконечном цикле, до тех пор, пока не встретится `-1`.

Когда встретится, сумма делится на число всех значений (исключая `-1`) и **частное** возвращается.

Так что, другими словами, я бы сказал, функция обходится с фрагментом стека как с массивом целочисленных значений, бесконечной длины.

Теперь нам легче понять почему в соглашениях о вызовах `cdecl` первый аргумент заталкивается в стек последним.

Потому что иначе будет невозможно найти первый аргумент, или, для функции вроде `printf()`, невозможно будет найти строку формата.

### Соглашения о вызовах на основе регистров

Наблюдательный читатель может спросить, что насчет тех соглашений о вызовах, где первые аргументы передаются в регистрах?

Посмотрим:

Листинг 3.59: Оптимизирующий MSVC 2012 x64

```

$SG3013 DB    '%d', 0aH, 00H

v$ = 8
arith_mean PROC
    mov     DWORD PTR [rsp+8], ecx    ; первый аргумент
    mov     QWORD PTR [rsp+16], rdx   ; второй аргумент
    mov     QWORD PTR [rsp+24], r8    ; третий аргумент
    mov     eax, ecx                 ; sum = первый аргумент
    lea    rcx, QWORD PTR v$[rsp+8] ; указатель на второй аргумент
    mov     QWORD PTR [rsp+32], r9    ; 4-й аргумент

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    mov     edx, DWORD PTR [rcx]      ; загрузить второй аргумент
    mov     r8d, 1                    ; count=1
    cmp     edx, -1                   ; второй аргумент равен -1?
    je     SHORT $LN8@arith_mean     ; если так, то выход
$LL3@arith_mean:
    add     eax, edx                  ; sum = sum + загруженный аргумент
    mov     edx, DWORD PTR [rcx+8]   ; загрузить следующий аргумент
    lea    rcx, QWORD PTR [rcx+8]   ; сдвинуть указатель, чтобы он
    указывал на аргумент за следующим
    inc     r8d                       ; count++
    cmp     edx, -1                   ; загруженный аргумент равен -1?
    jne    SHORT $LL3@arith_mean     ; перейти на начал цикла, если нет
$LN8@arith_mean:
; вычислить результат деления
    cdq
    idiv   r8d
    ret    0
arith_mean ENDP

main PROC
    sub     rsp, 56
    mov     edx, 2
    mov     DWORD PTR [rsp+40], -1
    mov     DWORD PTR [rsp+32], 15
    lea    r9d, QWORD PTR [rdx+8]
    lea    r8d, QWORD PTR [rdx+5]
    lea    ecx, QWORD PTR [rdx-1]
    call   arith_mean
    lea    rcx, OFFSET FLAT:$SG3013
    mov     edx, eax
    call   printf
    xor     eax, eax
    add     rsp, 56
    ret    0
main ENDP

```

Мы видим, что первые 4 аргумента передаются в регистрах и еще два — в стеке.

Функция `arith_mean()` в начале сохраняет эти 4 аргумента в *Shadow Space* и затем обходится с *Shadow Space* и стеком за ним как с единым непрерывным массивом!

Что насчет GCC? Тут немного неуклюже всё, потому что функция делится на две части: первая часть сохраняет регистры в «red zone», обрабатывает это пространство, а вторая часть функции обрабатывает стек:

#### Листинг 3.60: Оптимизирующий GCC 4.9.1 x64

```

arith_mean:
    lea    rax, [rsp+8]
    ; сохранить 6 входных регистров в
    ; red zone в локальном стеке
    mov     QWORD PTR [rsp-40], rsi
    mov     QWORD PTR [rsp-32], rdx

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov     QWORD PTR [rsp-16], r8
mov     QWORD PTR [rsp-24], rcx
mov     esi, 8
mov     QWORD PTR [rsp-64], rax
lea     rax, [rsp-48]
mov     QWORD PTR [rsp-8], r9
mov     DWORD PTR [rsp-72], 8
lea     rdx, [rsp+8]
mov     r8d, 1
mov     QWORD PTR [rsp-56], rax
jmp     .L5
.L7:
; обработать сохраненные аргументы
lea     rax, [rsp-48]
mov     ecx, esi
add     esi, 8
add     rcx, rax
mov     ecx, DWORD PTR [rcx]
cmp     ecx, -1
je     .L4
.L8:
add     edi, ecx
add     r8d, 1
.L5:
; решить, какую часть мы сейчас будем обрабатывать
; текущий номер аргумента меньше или равен 6?
cmp     esi, 47
jbe     .L7 ; нет, тогда обрабатываем сохраненные
аргументы
; обрабатываем аргументы из стека
mov     rcx, rdx
add     rdx, 8
mov     ecx, DWORD PTR [rcx]
cmp     ecx, -1
jne     .L8
.L4:
mov     eax, edi
cdq
idiv   r8d
ret

.LC1:
.string "%d\n"
main:
sub     rsp, 8
mov     edx, 7
mov     esi, 2
mov     edi, 1
mov     r9d, -1
mov     r8d, 15
mov     ecx, 10
xor     eax, eax
call   arith_mean
mov     esi, OFFSET FLAT:.LC1

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov    edx, eax
mov    edi, 1
xor    eax, eax
add    rsp, 8
jmp    __printf_chk

```

Кстати, похожее использование *Shadow Space* разбирается здесь: ?? (стр. ??).

### Используя указатель на первый аргумент ф-ции

Пример можно переписать без использования макроса `va_arg`:

```

#include <stdio.h>

int arith_mean(int v, ...)
{
    int *i=&v;
    int sum=*i, count=1;
    i++;

    while(1)
    {
        if ((*i)==-1) // terminator
            break;
        sum=sum+(*i);
        count++;
        i++;
    }

    return sum/count;
};

int main()
{
    printf ("%d\n", arith_mean (1, 2, 7, 10, 15, -1 /* terminator */));
    // test: https://www.wolframalpha.com/input/?i=mean(1,2,7,10,15)
};

```

Иными словами, если набор аргументов – это массив слов (32-битных или 64-битных), то мы просто перебираем элементы этого массива, начиная с первого.

### 3.15.2. Случай с функцией `vprintf()`

Многие программисты определяют свою собственную функцию для записи в лог, которая берет строку формата вида `printf()` + переменное количество аргументов.

Еще один популярный пример это функция `die()`, которая выводит некоторое сообщение и заканчивает работу.

Нам нужен какой-то способ запаковать входные аргументы неизвестного количества и передать их в функцию `printf()`.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Но как? Вот зачем нужны функции с «v» в названии.

Одна из них это *vprintf()*: она берет строку формата и указатель на переменную типа *va\_list*:

```
#include <stdlib.h>
#include <stdarg.h>

void die (const char * fmt, ...)
{
    va_list va;
    va_start (va, fmt);

    vprintf (fmt, va);
    exit(0);
};
```

При ближайшем рассмотрении, мы можем увидеть, что *va\_list* это указатель на массив.

Скомпилируем:

Листинг 3.61: Оптимизирующий MSVC 2010

```
_fmt$ = 8
_die PROC
    ; загрузить первый аргумент (строка формата)
    mov     ecx, DWORD PTR _fmt$[esp-4]
    ; установить указатель на второй аргумент
    lea    eax, DWORD PTR _fmt$[esp]
    push   eax                ; передать указатель
    push   ecx
    call   _vprintf
    add    esp, 8
    push   0
    call   _exit
$LN3@die:
    int    3
_die     ENDP
```

Мы видим, что всё что наша функция делает это просто берет указатель на аргументы, передает его в *vprintf()*, и эта функция работает с ним, как с бесконечным массивом аргументов!

Листинг 3.62: Оптимизирующий MSVC 2012 x64

```
fmt$ = 48
die PROC
    ; сохранить первые 4 аргумента в Shadow Space
    mov    QWORD PTR [rsp+8], rcx
    mov    QWORD PTR [rsp+16], rdx
    mov    QWORD PTR [rsp+24], r8
    mov    QWORD PTR [rsp+32], r9
    sub    rsp, 40
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

lea    rdx, QWORD PTR fmt$[rsp+8] ; передать указатель на первый
аргумент
; RCX здесь всё еще указывает на первый аргумент (строку формата)
φ-ции die()
; так что vprintf() возьмет его прямо из RCX
call   vprintf
xor    ecx, ecx
call   exit
int    3
die    ENDP

```

### 3.15.3. Случай с Pin

Интересно посмотреть, как некоторые ф-ции из **DBI!**<sup>16</sup> Pin берут на вход несколько аргументов:

```

INS_InsertPredicatedCall(
    ins, IPOINT_BEFORE, (AFUNPTR)RecordMemRead,
    IARG_INST_PTR,
    IARG_MEMORYOP_EA, memOp,
    IARG_END);

```

( pinatrace.cpp )

И вот как объявлена ф-ция `INS_InsertPredicatedCall()`:

```

extern VOID INS_InsertPredicatedCall(INS ins, IPOINT ipoint, AFUNPTR funptr,
    ↵ , ...);

```

( pin\_client.PH )

Следовательно, константы с именами начинающимися с `IARG_` это что-то вроде аргументов для ф-ции, которая обрабатывается внутри `INS_InsertPredicatedCall()`. Вы можете передавать столько аргументов, сколько нужно. Некоторые команды имеют дополнительные аргументы, некоторые другие — нет. Полный список аргументов: [https://software.intel.com/sites/landingpage/pintool/docs/58423/Pin/html/group\\_\\_INST\\_\\_ARGS.html](https://software.intel.com/sites/landingpage/pintool/docs/58423/Pin/html/group__INST__ARGS.html). И должен быть какой-то способ узнать, закончился ли список аргументов, так что список должен быть окончен при помощи константы `IARG_END`, без которой ф-ция будет (пытаться) обрабатывать случайный шум из локального стека, принимая его за дополнительные аргументы.

Также, в [Brian W. Kernighan, Rob Pike, *Practice of Programming*, (1999)] можно найти прекрасный пример ф-ций на Си/Си++, очень похожих на `pack/unpack`<sup>17</sup> в Python.

### 3.15.4. Эксплуатация строки формата

Есть популярная ошибка, писать `printf(string)` вместо `puts(string)` или `printf("%s", string)`. Если тот, кто пытается взломать систему удаленно, может указать

<sup>16</sup>**DBI!**

<sup>17</sup><https://docs.python.org/3/library/struct.html>

свою string, он/она может свалить процесс, или даже получить доступ к переменным в локальном стеке.

Посмотрите на это:

```
#include <stdio.h>

int main()
{
    char *s1="hello";
    char *s2="world";
    char buf[128];

    // do something mundane here
    strcpy (buf, s1);
    strcpy (buf, " ");
    strcpy (buf, s2);

    printf ("%s");
};
```

Нужно отметить, что у вызова printf() нет дополнительных аргументов кроме строки формата.

Теперь представим что это взломщик просунул строку %s как единственный аргумент последнего вызова printf(). Я компилирую это в GCC 5.4.0 на x86 Ubuntu, и итоговый исполняемый файл печатает строку «world» при запуске!

Если я включаю оптимизацию, printf() выдает какой-то мусор, хотя, вероятно, вызовы strcpy() были оптимизированы, и/или локальные переменные также. Также, результат будет другой для x64-кода, другого компилятора, **OS!**, и т. д.

Теперь, скажем, взломщик может передать эту строку в вызов printf(): %x %x %x %x. В моем случае, вывод это: «80485c6 b7751b48 1 0 80485c0» (это просто значения из локального стека). Как видите, есть значение 1 и 0, и еще некоторые указатели (первый, наверное, указатель на строку «world»). Так что если взломщик передаст строку %s %s %s %s %s, процесс упадет, потому что printf() считает 1 и/или 0 за указатель на строку, пытается читать символы оттуда, и терпит неудачу.

И даже хуже, в коде может быть sprintf (buf, string), где buf это буфер в локальном стеке с размером в 1024 байт или около того, взломщик может создать строку string таким образом, что buf будет переполнен, может быть даже в таком виде, что это приведет к исполнению кода.

Многое популярное ПО было (или даже до сих пор) уязвимо:

```
QuakeWorld went up, got to around 4000 users, then the master
server exploded.
Disrupter and cohorts are working on more robust code now.
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

If anyone did it on purpose, how about letting us know... (It wasn't all the people that tried %s as a name)

( .plan-файл Джона Кармака, 17-декабрь-1996<sup>18</sup> )

В наше время, почти все современные компиляторы предупреждают об этом.

Еще одна проблема это менее известный аргумент `printf() %n`: когда `printf()` доходит до него в строке формата, он пишет число выведенных символов в соответствующий аргумент: [stackoverflow.com](https://stackoverflow.com). Так, взломщик может затереть локальные переменные передавая в строке формата множество команд `%n`.

### 3.16. Обрезка строк

Весьма востребованная операция со строками — это удаление некоторых символов в начале и/или конце строки.

В этом примере, мы будем работать с функцией, удаляющей все символы перевода строки (**CR!**<sup>19</sup>/**LF!**<sup>20</sup>) в конце входной строки:

```
#include <stdio.h>
#include <string.h>

char* str_trim (char *s)
{
    char c;
    size_t str_len;

    // работать до тех пор, пока \r или \n находятся в конце строки
    // остановиться, если там какой-то другой символ, или если строка
    пустая
    // (на старте, или в результате наших действий)
    for (str_len=strlen(s); str_len>0 && (c=s[str_len-1]); str_len--)
    {
        if (c=='\r' || c=='\n')
            s[str_len-1]=0;
        else
            break;
    };
    return s;
};

int main()
{
    // тест

    // здесь применяется strdup() для копирования строк в сегмент данных,
```

<sup>18</sup>[https://github.com/ESWAT/john-carmack-plan-archive/blob/33ae52fdb46aa0d1abfed6fc7598233748541c0/by\\_day/johnc\\_plan\\_19961217.txt](https://github.com/ESWAT/john-carmack-plan-archive/blob/33ae52fdb46aa0d1abfed6fc7598233748541c0/by_day/johnc_plan_19961217.txt)

<sup>19</sup>**CR!**

<sup>20</sup>**LF!**

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

// потому что иначе процесс упадет в Linux,
// где текстовые строки располагаются в константном сегменте данных,
// и не могут модифицироваться.

printf ("%s\n", str_trim (strdup("")));
printf ("%s\n", str_trim (strdup("\n")));
printf ("%s\n", str_trim (strdup("\r")));
printf ("%s\n", str_trim (strdup("\n\r")));
printf ("%s\n", str_trim (strdup("\r\n")));
printf ("%s\n", str_trim (strdup("test1\r\n")));
printf ("%s\n", str_trim (strdup("test2\n\r")));
printf ("%s\n", str_trim (strdup("test3\n\r\n\r")));
printf ("%s\n", str_trim (strdup("test4\n")));
printf ("%s\n", str_trim (strdup("test5\r")));
printf ("%s\n", str_trim (strdup("test6\r\r\r")));
};

```

Входной аргумент всегда возвращается на выходе, это удобно, когда вам нужно объединять функции обработки строк в цепочки, как это сделано здесь в функции `main()`.

Вторая часть `for()` (`str_len > 0 && (c = s[str_len - 1])`) называется в Си/Си++ «short-circuit» (короткое замыкание) и это очень удобно: [Денис Юричев, *Заметки о языке программирования Си/Си++ 1.3.8*].

Компиляторы Си/Си++ гарантируют последовательное вычисление слева направо.

Так что если первое условие не истинно после вычисления, второе никогда не будет вычисляться.

### 3.16.1. x64: Оптимизирующий MSVC 2013

Листинг 3.63: Оптимизирующий MSVC 2013 x64

```

s$ = 8
str_trim PROC

; RCX это первый аргумент функции, и он всегда будет указывать на строку
mov     rdx, rcx
; это функция strlen() встроенная в код прямо здесь:
; установить RAX в 0xFFFFFFFFFFFFFFFF (-1)
or     rax, -1
$LL14@str_trim:
inc     rax
cmp     BYTE PTR [rcx+rax], 0
jne     $LL14@str_trim
; длина входной строки 0? тогда на выход:
test   rax, rax
je     $LN15@str_trim
; RAX содержит длину строки
dec     rcx
; RCX = s-1

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        mov     r8d, 1
        add     rcx, rax
; RCX = s-1+strlen(s), т.е., это адрес последнего символа в строке
        sub     r8, rdx
; R8 = 1-s
$LL6@str_trim:
; загрузить последний символ строки:
; перейти, если его код 13 или 10:
        movzx   eax, BYTE PTR [rcx]
        cmp     al, 13
        je     SHORT $LN2@str_trim
        cmp     al, 10
        jne    SHORT $LN15@str_trim
$LN2@str_trim:
; последний символ имеет код 13 или 10
; записываем ноль в этом месте:
        mov     BYTE PTR [rcx], 0
; декремент адреса последнего символа,
; так что он будет указывать на символ перед только что стертым:
        dec     rcx
        lea    rax, QWORD PTR [r8+rcx]
; RAX = 1 - s + адрес текущего последнего символа
; так мы определяем, достигли ли мы первого символа, и раз так, то нам нужно
остановиться
        test    rax, rax
        jne    SHORT $LL6@str_trim
$LN15@str_trim:
        mov     rax, rdx
        ret     0
str_trim ENDP

```

В начале, MSVC вставил тело функции `strlen()` прямо в код, потому что решил, что так будет быстрее чем обычная работа `strlen()` + время на вызов её и возврат из нее.

Это также называется *inlining*: ?? (стр. ??).

Первая инструкция функции `strlen()` вставленная здесь, это `OR RAX, 0xFFFFFFFFFFFFFFFF`. MSVC часто использует `OR` вместо `MOV RAX, 0xFFFFFFFFFFFFFFFF`, потому что опкод получается короче.

И конечно, это эквивалентно друг другу: все биты просто выставляются, а все выставленные биты это -1 в дополнительном коде (two's complement).

Кто-то мог бы спросить, зачем вообще нужно использовать число -1 в функции `strlen()`?

Вследствие оптимизации, конечно. Вот что сделал MSVC:

Листинг 3.64: Вставленная `strlen()` сгенерированная MSVC 2013 x64

```

; RCX = указатель на входную строку
; RAX = текущая длина строки
        or     rax, -1
label:
        inc    rax

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    cmp    BYTE PTR [rcx+rax], 0
    jne    SHORT label
; RAX = длина строки

```

Попробуйте написать короче, если хотите инициализировать счетчик нулем!

Ну, например:

#### Листинг 3.65: Наша версия strlen()

```

; RCX = указатель на входную строку
; RAX = текущая длина строки
xor     rax, rax
label:
    cmp    byte ptr [rcx+rax], 0
    jz     exit
    inc   rax
    jmp   label
exit:
; RAX = длина строки

```

Не получилось. Нам придется вводить дополнительную инструкцию JMP!

Что сделал MSVC 2013, так это передвинул инструкцию INC в место перед загрузкой символа.

Если самый первый символ — нулевой, всё нормально, RAX содержит 0 в этот момент, так что итоговая длина строки будет 0.

Остальную часть функции проще понять.

### 3.16.2. x64: Неоптимизирующий GCC 4.9.1

```

str_trim:
    push   rbp
    mov    rbp, rsp
    sub   rsp, 32
    mov    QWORD PTR [rbp-24], rdi
; здесь начинается первая часть for()
    mov    rax, QWORD PTR [rbp-24]
    mov    rdi, rax
    call  strlen
    mov    QWORD PTR [rbp-8], rax    ; str_len
; здесь заканчивается первая часть for()
    jmp   .L2
; здесь начинается тело for()
.L5:
    cmp    BYTE PTR [rbp-9], 13    ; c=='\r'?
    je     .L3
    cmp    BYTE PTR [rbp-9], 10    ; c=='\n'?
    jne    .L4
.L3:
    mov    rax, QWORD PTR [rbp-8]    ; str_len
    lea   rdx, [rax-1]              ; EDX=str_len-1

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    mov    rax, QWORD PTR [rbp-24] ; s
    add    rax, rdx                ; RAX=s+str_len-1
    mov    BYTE PTR [rax], 0      ; s[str_len-1]=0
; тело for() заканчивается здесь
; здесь начинается третья часть for()
    sub    QWORD PTR [rbp-8], 1   ; str_len--
; здесь заканчивается третья часть for()
.L2:
; здесь начинается вторая часть for()
    cmp    QWORD PTR [rbp-8], 0   ; str_len==0?
    je     .L4                    ; тогда на выход
; проверить второе условие, и загрузить "с"
    mov    rax, QWORD PTR [rbp-8] ; RAX=str_len
    lea   rdx, [rax-1]           ; RDX=str_len-1
    mov    rax, QWORD PTR [rbp-24] ; RAX=s
    add    rax, rdx              ; RAX=s+str_len-1
    movzx eax, BYTE PTR [rax]    ; AL=s[str_len-1]
    mov    BYTE PTR [rbp-9], al   ; записать загруженный символ в "с"
    cmp    BYTE PTR [rbp-9], 0   ; это ноль?
    jne   .L5                    ; да? тогда на выход
; здесь заканчивается вторая часть for()
.L4:
; возврат "s"
    mov    rax, QWORD PTR [rbp-24]
    leave
    ret

```

Комментарии автора. После исполнения `strlen()`, управление передается на метку `L2`, и там проверяются два выражения, одно после другого.

Второе никогда не будет проверяться, если первое выражение не истинно (`str_len==0`) (это «short-circuit»).

Теперь посмотрим на эту функцию в коротком виде:

- Первая часть `for()` (вызов `strlen()`)
- `goto L2`
- `L5`: Тело `for()`. переход на выход, если нужно
- Третья часть `for()` (декремент `str_len`)
- `L2`: Вторая часть `for()`: проверить первое выражение, затем второе. переход на начало тела цикла, или выход.
- `L4`: // выход
- `return s`

### 3.16.3. x64: Оптимизирующий GCC 4.9.1

```

str_trim:
    push   rbx
    mov    rbx, rdi

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; RBX всегда будет "s"
    call    strlen
; проверить на str_len==0 и выйти, если это так
    test   rax, rax
    je     .L9
    lea   rdx, [rax-1]
; RDX всегда будет содержать значение str_len-1, но не str_len
; так что RDX будет скорее индексом буфера
    lea   rsi, [rbx+rdx]      ; RSI=s+str_len-1
    movzx ecx, BYTE PTR [rsi] ; загрузить символ
    test  cl, cl
    je    .L9                ; выйти, если это ноль
    cmp   cl, 10
    je    .L4
    cmp   cl, 13             ; выйти, если это не '\n' и не '\r'
    jne   .L9
.L4:
; это странная инструкция. нам здесь нужно RSI=s-1
; это можно сделать, используя MOV RSI, EBX / DEC RSI
; но это две инструкции между одной
    sub   rsi, rax
; RSI = s+str_len-1-str_len = s-1
; начало главного цикла
.L12:
    test  rdx, rdx
; записать ноль по адресу s-1+str_len-1+1 = s-1+str_len = s+str_len-1
    mov   BYTE PTR [rsi+1+rdx], 0
; проверка на str_len-1==0. выход, если да.
    je    .L9
    sub   rdx, 1             ; эквивалент str_len--
; загрузить следующий символ по адресу s+str_len-1
    movzx ecx, BYTE PTR [rbx+rdx]
    test  cl, cl             ; это ноль? тогда выход
    je    .L9
    cmp   cl, 10             ; это '\n'?
    je    .L12
    cmp   cl, 13             ; это '\r'?
    je    .L12
.L9:
; возврат "s"
    mov   rax, rbx
    pop   rbx
    ret

```

Тут более сложный результат. Код перед циклом выполняется только один раз, но также содержит проверку символов **CR! / LF!**

Зачем нужна это дублирование кода?

Обычная реализация главного цикла это, наверное, такая:

- (начало цикла) проверить символы **CR! / LF!**, принять решения
- записать нулевой символ

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Но GCC решил поменять местами эти два шага. Конечно, шаг *записать нулевой символ* не может быть первым, так что нужна еще одна проверка:

- обработать первый символ. сравнить его с **CR!/LF!**, выйти если символ не равен **CR!/LF!**
- (начало цикла) записать нулевой символ
- проверить символы **CR!/LF!**, принять решения

Теперь основной цикл очень короткий, а это очень хорошо для современных процессоров.

Код не использует переменную `str_len`, но `str_len-1`.

Так что это больше похоже на индекс в буфере. Должно быть, GCC заметил, что выражение `str_len-1` используется дважды.

Так что будет лучше выделить переменную, которая всегда содержит значение равное текущей длине строки минус 1, и уменьшать его на 1 (это тот же эффект, что и уменьшать переменную `str_len`).

### 3.16.4. ARM64: Неоптимизирующий GCC (Linaro) 4.9

Реализация простая и прямолинейная:

Листинг 3.66: Неоптимизирующий GCC (Linaro) 4.9

```
str_trim:
    stp    x29, x30, [sp, -48]!
    add   x29, sp, 0
    str   x0, [x29,24] ; скопировать входной аргумент в локальный стек
    ldr   x0, [x29,24] ; s
    bl   strlen
    str   x0, [x29,40] ; переменная str_len в локальном стеке
    b    .L2
; начало главного цикла
.L5:
    ldrb  w0, [x29,39]
; W0=c
    cmp   w0, 13      ; это '\r'?
    beq   .L3
    ldrb  w0, [x29,39]
; W0=c
    cmp   w0, 10      ; это '\n'?
    bne   .L4        ; перейти в конец, если нет
.L3:
    ldr   x0, [x29,40]
; X0=str_len
    sub   x0, x0, #1
; X0=str_len-1
    ldr   x1, [x29,24]
; X1=s
    add   x0, x1, x0
; X0=s+str_len-1
    strb  wzr, [x0]   ; записать байт на s+str_len-1
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; декремент str_len:
    ldr    x0, [x29,40]
; X0=str_len
    sub    x0, x0, #1
; X0=str_len-1
    str    x0, [x29,40]
; сохранить X0 (или str_len-1) в локальном стеке
.L2:
    ldr    x0, [x29,40]
; str_len==0?
    cmp    x0, xzr
; перейти на выход, если да
    beq    .L4
    ldr    x0, [x29,40]
; X0=str_len
    sub    x0, x0, #1
; X0=str_len-1
    ldr    x1, [x29,24]
; X1=s
    add    x0, x1, x0
; X0=s+str_len-1
; загрузить байт по адресу s+str_len-1 в W0
    ldrb   w0, [x0]
    strb   w0, [x29,39] ; сохранить загруженный байт в "с"
    ldrb   w0, [x29,39] ; перезагрузить его
; это нулевой байт?
    cmp    w0, wzr
; перейти на конец, если это ноль, или на L5, если нет
    bne    .L5
.L4:
; возврат s
    ldr    x0, [x29,24]
    ldp    x29, x30, [sp], 48
    ret

```

### 3.16.5. ARM64: Оптимизирующий GCC (Linaro) 4.9

Это более продвинутая оптимизация. Первый символ загружается в самом начале и сравнивается с 10 (символ **LF!**).

Символы также загружаются и в главном цикле, для символов после первого.

Это в каком смысле похоже на этот пример: ?? (стр. ??).

Листинг 3.67: Оптимизирующий GCC (Linaro) 4.9

```

str_trim:
    stp    x29, x30, [sp, -32]!
    add    x29, sp, 0
    str    x19, [sp,16]
    mov    x19, x0
; X19 всегда будет содержать значение "s"
    bl    strlen

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; X0=str_len
    cbz    x0, .L9          ; перейти на L9 (выход), если str_len==0
    sub    x1, x0, #1
; X1=X0-1=str_len-1
    add    x3, x19, x1
; X3=X19+X1=s+str_len-1
    ldrb   w2, [x19,x1]    ; загрузить байт по адресу X19+X1=s+str_len-1
; W2=загруженный символ
    cbz    w2, .L9          ; это ноль? тогда перейти на выход
    cmp    w2, 10           ; это '\n'?
    bne    .L15
.L12:
; тело главного цикла. загруженный символ в этот момент всегда 10 или 13!
    sub    x2, x1, x0
; X2=X1-X0=str_len-1-str_len=-1
    add    x2, x3, x2
; X2=X3+X2=s+str_len-1+(-1)=s+str_len-2
    strb   wzr, [x2,1]     ; записать нулевой байт по адресу
    s+str_len-2+1=s+str_len-1
    cbz    x1, .L9          ; str_len-1==0? перейти на выход, если это так
    sub    x1, x1, #1      ; str_len--
    ldrb   w2, [x19,x1]    ; загрузить следующий символ по адресу
    X19+X1=s+str_len-1
    cmp    w2, 10           ; это '\n'?
    cbz    w2, .L9          ; перейти на выход, если это ноль
    beq    .L12            ; перейти на начало цикла, если это '\n'
.L15:
    cmp    w2, 13           ; это '\r'?
    beq    .L12            ; да, перейти на начало тела цикла
.L9:
; возврат "s"
    mov    x0, x19
    ldr    x19, [sp,16]
    ldp    x29, x30, [sp], 32
    ret

```

### 3.16.6. ARM: Оптимизирующий Keil 6/2013 (Режим ARM)

И снова, компилятор пользуется условными инструкциями в режиме ARM, поэтому код более компактный.

Листинг 3.68: Оптимизирующий Keil 6/2013 (Режим ARM)

```

str_trim PROC
    PUSH   {r4,lr}
; R0=s
    MOV    r4,r0
; R4=s
    BL     strlen          ; strlen() берет значение "s" из R0
; R0=str_len
    MOV    r3,#0
; R3 всегда будет содержать 0
|L0.16|

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

CMP     r0,#0      ; str_len==0?
ADDNE   r2,r4,r0   ; (если str_len!=0) R2=R4+R0=s+str_len
LDRBNE  r1,[r2,#-1] ; (если str_len!=0) R1=загрузить байт по адресу
R2-1=s+str_len-1
CMPNE   r1,#0     ; (если str_len!=0) сравнить загруженный байт с
0
BEQ     |L0.56|    ; перейти на выход, если str_len==0 или если
загруженный байт - это 0
CMP     r1,#0xd    ; загруженный байт - это '\r'?
CMPNE   r1,#0xa    ;
(если загруженный байт - это не '\r') загруженный байт - это '\r'?
SUBEQ   r0,r0,#1   ;
(если загруженный байт - это '\r' или '\n') R0-- или str_len--
STRBEQ  r3,[r2,#-1] ; (если загруженный байт - это '\r' или '\n')
записать R3 (ноль) по адресу R2-1=s+str_len-1
BEQ     |L0.16|    ;
перейти на начало цикла, если загруженный байт был '\r' или '\n'
|L0.56|
; возврат "s"
MOV     r0,r4
POP     {r4,pc}
ENDP

```

### 3.16.7. ARM: Оптимизирующий Keil 6/2013 (Режим Thumb)

В режиме Thumb куда меньше условных инструкций, так что код более простой.

Но здесь есть одна странность со сдвигами на 0x20 и 0x1F (строки 22 и 23).

Почему компилятор Keil сделал так? Честно говоря, трудно сказать. Возможно, это выверт процесса оптимизации компилятора.

Тем не менее, код будет работать корректно.

Листинг 3.69: Оптимизирующий Keil 6/2013 (Режим Thumb)

```

1 str_trim PROC
2     PUSH     {r4,lr}
3     MOVS    r4,r0
4     ; R4=s
5     BL      strlen      ; strlen() берет значение "s" из R0
6     ; R0=str_len
7     MOVS    r3,#0
8     ; R3 всегда будет содержать 0
9     B       |L0.24|
10    |L0.12|
11    CMP     r1,#0xd     ; загруженный байт - это '\r'?
12    BEQ     |L0.20|
13    CMP     r1,#0xa    ; загруженный байт - это '\n'?
14    BNE    |L0.38|    ; перейти на выход, если нет
15    |L0.20|
16    SUBS    r0,r0,#1   ; R0-- или str_len--
17    STRB    r3,[r2,#0x1f] ; записать 0 по адресу
R2+0x1f=s+str_len-0x20+0x1f=s+str_len-1

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

18 |L0.24|
19     CMP     r0,#0           ; str_len==0?
20     BEQ     |L0.38|        ; да? тогда перейти на выход
21     ADDS    r2,r4,r0       ; R2=R4+R0=s+str_len
22     SUBS    r2,r2,#0x20    ; R2=R2-0x20=s+str_len-0x20
23     LDRB    r1,[r2,#0x1f] ; загрузить байт по адресу
                          R2+0x1F=s+str_len-0x20+0x1F=s+str_len-1 в R1
24     CMP     r1,#0         ; загруженный байт - это 0?
25     BNE     |L0.12|       ; перейти на начало цикла, если это не 0
26 |L0.38|
27 ; возврат "s"
28     MOVS    r0,r4
29     POP     {r4,pc}
30     ENDP

```

### 3.16.8. MIPS

Листинг 3.70: Оптимизирующий GCC 4.4.5 (IDA)

```

str_trim:
; IDA не в курсе об именах переменных, мы присвоили их сами:
saved_GP      = -0x10
saved_S0      = -8
saved_RA      = -4

        lui    $gp, (__gnu_local_gp >> 16)
        addiu  $sp, -0x20
        la    $gp, (__gnu_local_gp & 0xFFFF)
        sw    $ra, 0x20+saved_RA($sp)
        sw    $s0, 0x20+saved_S0($sp)
        sw    $gp, 0x20+saved_GP($sp)
; вызов strlen(). адрес входной строки всё еще в $a0, strlen() возьмет его
  оттуда:
        lw    $t9, (strlen & 0xFFFF)($gp)
        or    $at, $zero ; load delay slot, NOP
        jalr  $t9
; адрес входной строки всё еще в $a0, переложить его в $s0:
        move  $s0, $a0 ; branch delay slot
; результат strlen() (т.е., длина строки) теперь в $v0
; перейти на выход, если $v0==0 (т.е., если длина строки это 0):
        beqz  $v0, exit
        or    $at, $zero ; branch delay slot, NOP
        addiu $al, $v0, -1
; $al = $v0-1 = strlen-1
        addu  $al, $s0, $al
; $al = адрес входной строки + $al = s+strlen-1
; загрузить байт по адресу $al:
        lb    $a0, 0($al)
        or    $at, $zero ; load delay slot, NOP
; загруженный байт - это ноль? перейти на выход, если это так:
        beqz  $a0, exit
        or    $at, $zero ; branch delay slot, NOP
        addiu $v1, $v0, -2

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; $v1 = str_len-2
    addu    $v1, $s0, $v1
; $v1 = $s0+$v1 = s+str_len-2
    li     $a2, 0xD
; пропустить тело цикла:
    b      loc_6C
    li     $a3, 0xA ; branch delay slot
loc_5C:
; загрузить следующий байт из памяти в $a0:
    lb     $a0, 0($v1)
    move   $a1, $v1
; $a1=s+str_len-2
; перейти на выход, если загруженный байт - это ноль:
    beqz   $a0, exit
; декремент str_len:
    addiu  $v1, -1 ; branch delay slot
loc_6C:
; в этот момент, $a0=загруженный байт, $a2=0xD (символ CR) и $a3=0xA (символ
LF)
; загруженный байт - это CR? тогда перейти на loc_7C:
    beq    $a0, $a2, loc_7C
    addiu  $v0, -1 ; branch delay slot
; загруженный байт - это LF? перейти на выход, если это не LF:
    bne    $a0, $a3, exit
    or     $at, $zero ; branch delay slot, NOP
loc_7C:
; загруженный байт в этот момент это CR
; перейти на loc_5c (начало тела цикла) если str_len (в $v0) не ноль:
    bnez   $v0, loc_5C
; одновременно с этим, записать ноль в этом месте памяти:
    sb     $zero, 0($a1) ; branch delay slot
; метка "exit" была так названа мною:
exit:
    lw     $ra, 0x20+saved_RA($sp)
    move   $v0, $s0
    lw     $s0, 0x20+saved_S0($sp)
    jr     $ra
    addiu  $sp, 0x20 ; branch delay slot

```

Регистры с префиксом S- называются «saved temporaries», так что, значение \$S0 сохраняется в локальном стеке и восстанавливается во время выхода.

### 3.17. Функция toupper()

Еще одна очень востребованная функция конвертирует символ из строчного в заглавный, если нужно:

```

char toupper (char c)
{
    if(c>='a' && c<='z')
        return c-'a'+'A';
    else

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    return c;
}

```

Выражение 'a'+ 'A' оставлено в исходном коде для удобства чтения, конечно, оно оптимизируется

<sup>21</sup>.

**ASCII!**-код символа «a» это 97 (или 0x61), и 65 (или 0x41) для символа «A».

Разница (или расстояние) между ними в **ASCII!**-таблице это 32 (или 0x20).

Для лучшего понимания, читатель может посмотреть на стандартную 7-битную таблицу **ASCII!**:

Characters in the coded character set ascii.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x	C-@	C-a	C-b	C-c	C-d	C-e	C-f	C-g	C-h	TAB	C-j	C-k	C-l	RET	C-n	C-o
1x	C-p	C-q	C-r	C-s	C-t	C-u	C-v	C-w	C-x	C-y	C-z	ESC	C-\	C-]	C-^	C-_
2x	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Рис. 3.3: 7-битная таблица **ASCII!** в Emacs

### 3.17.1. x64

#### Две операции сравнения

Неоптимизирующий MSVC прямолинеен: код проверяет, находится ли входной символ в интервале [97..122] (или в интервале ['a'..'z']) и вычитает 32 в таком случае.

Имеется также небольшой артефакт компилятора:

Листинг 3.71: Неоптимизирующий MSVC 2013 (x64)

```

1  c$ = 8
2  toupper PROC
3      mov     BYTE PTR [rsp+8], cl
4      movsx  eax, BYTE PTR c$[rsp]
5      cmp     eax, 97
6      jl     SHORT $LN2@toupper
7      movsx  eax, BYTE PTR c$[rsp]
8      cmp     eax, 122
9      jg     SHORT $LN2@toupper
10     movsx  eax, BYTE PTR c$[rsp]
11     sub     eax, 32

```

<sup>21</sup>Впрочем, если быть дотошным, вполне могут до сих пор существовать компиляторы, которые не оптимизируют подобное и оставляют в коде.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

12     jmp     SHORT $LN3@toupper
13     jmp     SHORT $LN1@toupper      ; артефакт компилятора
14 $LN2@toupper:
15     movzx  eax, BYTE PTR c$[rsp]    ; необязательное приведение типов
16 $LN1@toupper:
17 $LN3@toupper:
18     ret     0                       ; артефакт компилятора
19 toupper ENDP

```

Важно отметить что (на строке 3) входной байт загружается в 64-битный слот локального стека.

Все остальные биты ([8..63]) не трогаются, т.е. содержат случайный шум (вы можете увидеть его в отладчике).

Все инструкции работают только с байтами, так что всё нормально.

Последняя инструкция MOVZX на строке 15 берет байт из локального стека и расширяет его до 32-битного *int*, дополняя нулями.

Неоптимизирующий GCC делает почти то же самое:

Листинг 3.72: Неоптимизирующий GCC 4.9 (x64)

```

toupper:
    push    rbp
    mov     rbp, rsp
    mov     eax, edi
    mov     BYTE PTR [rbp-4], al
    cmp     BYTE PTR [rbp-4], 96
    jle    .L2
    cmp     BYTE PTR [rbp-4], 122
    jg     .L2
    movzx  eax, BYTE PTR [rbp-4]
    sub    eax, 32
    jmp    .L3
.L2:
    movzx  eax, BYTE PTR [rbp-4]
.L3:
    pop    rbp
    ret

```

### Одна операция сравнения

Оптимизирующий MSVC работает лучше, он генерирует только одну операцию сравнения:

Листинг 3.73: Оптимизирующий MSVC 2013 (x64)

```

toupper PROC
    lea    eax, DWORD PTR [rcx-97]
    cmp    al, 25
    ja     SHORT $LN2@toupper
    movsx  eax, cl

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    sub    eax, 32
    ret    0
$LN2@toupper:
    movzx  eax, cl
    ret    0
toupper ENDP

```

Уже было описано, как можно заменить две операции сравнения на одну: ?? (стр. ??).

Мы бы переписал это на Си/Си++ так:

```

int tmp=c-97;

if (tmp>25)
    return c;
else
    return c-32;

```

Переменная *tmp* должна быть знаковая.

При помощи этого, имеем две операции вычитания в случае конверсии плюс одну операцию сравнения.

В то время как оригинальный алгоритм использует две операции сравнения плюс одну операцию вычитания.

Оптимизирующий GCC даже лучше, он избавился от переходов (а это хорошо: ?? (стр. ??)) используя инструкцию `CMOVcc`:

Листинг 3.74: Оптимизирующий GCC 4.9 (x64)

```

1  toupper:
2      lea    edx, [rdi-97] ; 0x61
3      lea    eax, [rdi-32] ; 0x20
4      cmp    dl, 25
5      cmova  eax, edi
6      ret

```

На строке 3 код готовит уже сконvertированное значение заранее, как если бы конверсия всегда происходила.

На строке 5 это значение в EAX заменяется нетронутым входным значением, если конверсия не нужна. И тогда это значение (конечно, неверное), просто выбрасывается.

Вычитание с упреждением это цена, которую компилятор платит за отсутствие условных переходов.

### 3.17.2. ARM

Оптимизирующий Keil для режима ARM также генерирует только одну операцию сравнения:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Листинг 3.75: Оптимизирующий Keil 6/2013 (Режим ARM)

```

toupper PROC
    SUB     r1,r0,#0x61
    CMP     r1,#0x19
    SUBLS   r0,r0,#0x20
    ANDLS   r0,r0,#0xff
    BX      lr
    ENDP

```

SUBLS и ANDLS исполняются только если значение R1 меньше чем 0x19 (или равно). Они и делают конверсию.

Оптимизирующий Keil для режима Thumb также генерирует только одну операцию сравнения:

Листинг 3.76: Оптимизирующий Keil 6/2013 (Режим Thumb)

```

toupper PROC
    MOVS    r1,r0
    SUBS    r1,r1,#0x61
    CMP     r1,#0x19
    BHI     |L0.14|
    SUBS    r0,r0,#0x20
    LSLS    r0,r0,#24
    LSRS    r0,r0,#24
|L0.14|
    BX      lr
    ENDP

```

Последние две инструкции LSLS и LSRS работают как AND reg, 0xFF: это аналог Си/Си++-выражения  $(i \ll 24) \gg 24$ .

Очевидно, Keil для режима Thumb решил, что две 2-байтных инструкции это короче чем код, загружающий константу 0xFF плюс инструкция AND.

## GCC для ARM64

Листинг 3.77: Неоптимизирующий GCC 4.9 (ARM64)

```

toupper:
    sub     sp, sp, #16
    strb   w0, [sp,15]
    ldrb   w0, [sp,15]
    cmp    w0, 96
    bls    .L2
    ldrb   w0, [sp,15]
    cmp    w0, 122
    bhi    .L2
    ldrb   w0, [sp,15]
    sub    w0, w0, #32
    uxtb   w0, w0
    b      .L3
.L2:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.L3:   ldrb    w0, [sp,15]
      add    sp, sp, 16
      ret

```

Листинг 3.78: Оптимизирующий GCC 4.9 (ARM64)

```

toupper:
    uxtb    w0, w0
    sub    w1, w0, #97
    uxtb    w1, w1
    cmp    w1, 25
    bhi    .L2
    sub    w0, w0, #32
    uxtb    w0, w0
.L2:
    ret

```

### 3.17.3. Используя битовые операции

Учитывая тот факт, что 5-й бит (считая с 0-го) всегда присутствует после проверки, вычитание его это просто сброс этого единственного бита, но точно такого же эффекта можно достичь при помощи обычного применения операции “И”.

И даже проще, с исключаящим ИЛИ:

```

char toupper (char c)
{
    if(c>='a' && c<='z')
        return c^0x20;
    else
        return c;
}

```

Код близок к тому, что сгенерировал оптимизирующий GCC для предыдущего примера (?? (стр. ??)):

Листинг 3.79: Оптимизирующий GCC 5.4 (x86)

```

toupper:
    mov     edx, DWORD PTR [esp+4]
    lea    ecx, [edx-97]
    mov    eax, edx
    xor    eax, 32
    cmp    cl, 25
    cmova  eax, edx
    ret

```

...но используется XOR вместо SUB.

Переворачивание 5-го бита это просто перемещение *курсора* в таблице **ASCII!** вверх/вниз на 2 ряда.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



Некоторые люди говорят, что буквы нижнего/верхнего регистра были расставлены в **ASCII**-таблице таким манером намеренно, потому что:

Very old keyboards used to do Shift just by toggling the 32 or 16 bit, depending on the key; this is why the relationship between small and capital letters in ASCII is so regular, and the relationship between numbers and symbols, and some pairs of symbols, is sort of regular if you squint at it.

( Eric S. Raymond, <http://www.catb.org/esr/faqs/things-every-hacker-once-knew/> )

Следовательно, мы можем написать такой фрагмент кода, который просто меняет регистр букв:

```
#include <stdio.h>

char flip (char c)
{
    if((c>='a' && c<='z') || (c>='A' && c<='Z'))
        return c^0x20;
    else
        return c;
}

int main()
{
    // выдаст "hELLO, WORLD!"
    for (char *s="Hello, world!"; *s; s++)
        printf ("%c", flip(*s));
};
```

### 3.17.4. Итог

Все эти оптимизации компиляторов очень популярны в наше время и практикующий reverse engineer обычно часто видит такие варианты кода.

## 3.18. Обфускация

Обфускация это попытка спрятать код (или его значение) от reverse engineer-а.

### 3.18.1. Текстовые строки

Как мы знаем из (?? (стр. ??)) текстовые строки могут быть крайне полезны. Знающие об этом программисты могут попытаться их спрятать так, чтобы их не было видно в **IDA!** или любом шестнадцатеричном редакторе.

Вот простейший метод.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Вот как строка может быть сконструирована:

```
mov    byte ptr [ebx], 'h'
mov    byte ptr [ebx+1], 'e'
mov    byte ptr [ebx+2], 'l'
mov    byte ptr [ebx+3], 'l'
mov    byte ptr [ebx+4], 'o'
mov    byte ptr [ebx+5], ' '
mov    byte ptr [ebx+6], 'w'
mov    byte ptr [ebx+7], 'o'
mov    byte ptr [ebx+8], 'r'
mov    byte ptr [ebx+9], 'l'
mov    byte ptr [ebx+10], 'd'
```

Строка также может сравниваться с другой:

```
mov    ebx, offset username
cmp    byte ptr [ebx], 'j'
jnz    fail
cmp    byte ptr [ebx+1], 'o'
jnz    fail
cmp    byte ptr [ebx+2], 'h'
jnz    fail
cmp    byte ptr [ebx+3], 'n'
jnz    fail
jz     it_is_john
```

В обоих случаях, эти строки нельзя так просто найти в шестнадцатеричном редакторе.

Кстати, точно также со строками можно работать в тех случаях, когда строку нельзя разместить в сегменте данных, например, в **PIC!**<sup>22</sup>, или в шелл-коде.

Еще метод с использованием функции `sprintf()` для конструирования:

```
sprintf(buf, "%5%c%c%c%c", "hel", 'l', "o w", 'o', "rld");
```

Код выглядит ужасно, но как простейшая мера для анти-реверсинга, это может помочь.

Текстовые строки могут также присутствовать в зашифрованном виде, в таком случае, их использование будет предварять вызов функции для дешифровки.

Например: ?? (стр. ??).

### 3.18.2. Исполняемый код

#### Вставка мусора

Обфускация исполняемого кода — это вставка случайного мусора (между настоящим кодом), который исполняется, но не делает ничего полезного.

Просто пример:

<sup>22</sup>**PIC!**

Листинг 3.80: оригинальный код

```
add    eax, ebx
mul    ecx
```

Листинг 3.81: obfuscated code

```
xor    esi, 011223344h ; мусор
add    esi, eax        ; мусор
add    eax, ebx
mov    edx, eax        ; мусор
shl    edx, 4          ; мусор
mul    ecx
xor    esi, ecx        ; мусор
```

Здесь код-мусор использует регистры, которые не используются в настоящем коде (ESI и EDX). Впрочем, промежуточные результаты полученные при исполнении настоящего кода вполне могут использоваться кодом-мусором для бóльшей путаницы — почему нет?

### Замена инструкций на раздутые эквиваленты

- MOV op1, op2 может быть заменена на пару PUSH op2 / POP op1.
- JMP label может быть заменена на пару PUSH label / RET. **IDA!** не покажет ссылок на эту метку.
- CALL label может быть заменена на следующую тройку инструкций: PUSH label\_after\_CALL\_instruction / PUSH label / RET.
- PUSH op также можно заменить на пару инструкций: SUB ESP, 4 (или 8) / MOV [ESP], op.

### Всегда исполняющийся/никогда не исполняющийся код

Если разработчик уверен, что в ESI всегда будет 0 в этом месте:

```
mov    esi, 1
...    ; какой-то не трогающий ESI код
dec    esi
...    ; какой-то не трогающий ESI код
cmp    esi, 0
jz     real_code
; фальшивый багаж
real_code:
```

Reverse engineer-у понадобится какое-то время чтобы с этим разобраться.

Это также называется *opaque predicate*.

Еще один пример (и снова разработчик уверен, что ESI — всегда ноль):

```
; ESI=0
add    eax, ebx        ; реальный код
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mul    ecx          ; реальный код
add    eax, esi     ; oраque predicate. вместо ADD тут может быть XOR,
                    AND или SHL, и т.д.

```

### Сделать побольше путаницы

```

instruction 1
instruction 2
instruction 3

```

Можно заменить на:

```

begin:      jmp    ins1_label

ins2_label: instruction 2
            jmp    ins3_label

ins3_label: instruction 3
            jmp    exit:

ins1_label: instruction 1
            jmp    ins2_label

exit:

```

### Использование косвенных указателей

```

dummy_data1 db    100h dup (0)
message1    db    'hello world',0

dummy_data2 db    200h dup (0)
message2    db    'another message',0

func        proc
            ...
            mov    eax, offset dummy_data1 ; PE or ELF reloc here
            add    eax, 100h
            push   eax
            call   dump_string
            ...
            mov    eax, offset dummy_data2 ; PE or ELF reloc here
            add    eax, 200h
            push   eax
            call   dump_string
            ...
func        endp

```

**IDA!** покажет ссылки на `dummy_data1` и `dummy_data2`, но не на сами текстовые строки.

К глобальным переменным и даже функциям можно обращаться так же.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

### 3.18.3. Виртуальная машина / псевдо-код

Программист может также создать свой собственный **PL!** или **ISA!** и интерпретатор для него.

(Как версии Visual Basic перед 5.0, .NET или Java-машины.)

Reverse engineer-у придется потратить какое-то время для понимания деталей всех инструкций в **ISA!**. Ему также возможно придется писать что-то вроде дизассемблера/декомпилятора.

### 3.18.4. Еще кое-что

Моя попытка (хотя и слабая) пропатчить компилятор Tiny C чтобы он выдавал обфусцированный код: <http://blog.yurichev.com/node/58>.

Использование инструкции MOV для действительно сложных вещей: [Stephen Dolan, *mov is Turing-complete*, (2013)]<sup>23</sup>.

### 3.18.5. Упражнение

- <http://challenges.re/29>

## 3.19. Си++

### 3.19.1. Классы

#### Простой пример

Внутреннее представление классов в Си++ почти такое же, как и представление структур.

Давайте попробуем простой пример с двумя переменными, двумя конструкторами и одним методом:

```
#include <stdio.h>

class c
{
private:
    int v1;
    int v2;
public:
    c() // конструктор по умолчанию
    {
        v1=667;
        v2=999;
    };

    c(int a, int b) // конструктор
```

<sup>23</sup>Также доступно здесь: <http://www.cl.cam.ac.uk/~sd601/papers/mov.pdf>

```

    {
        v1=a;
        v2=b;
    };

    void dump()
    {
        printf ("%d; %d\n", v1, v2);
    };
};

int main()
{
    class c c1;
    class c c2(5,6);

    c1.dump();
    c2.dump();

    return 0;
};

```

**MSVC: x86**

Вот как выглядит main() на ассемблере:

Листинг 3.82: MSVC

```

_c2$ = -16 ; size = 8
_c1$ = -8  ; size = 8
_main PROC
    push ebp
    mov  ebp, esp
    sub  esp, 16
    lea  ecx, DWORD PTR _c1$[ebp]
    call ??0c@@QAE@XZ ; c::c
    push 6
    push 5
    lea  ecx, DWORD PTR _c2$[ebp]
    call ??0c@@QAE@HH@Z ; c::c
    lea  ecx, DWORD PTR _c1$[ebp]
    call ?dump@c@@QAE@XZ ; c::dump
    lea  ecx, DWORD PTR _c2$[ebp]
    call ?dump@c@@QAE@XZ ; c::dump
    xor  eax, eax
    mov  esp, ebp
    pop  ebp
    ret  0
_main ENDP

```

Вот что происходит. Под каждый экземпляр класса *c* выделяется по 8 байт, столько же, сколько нужно для хранения двух переменных.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Для *c1* вызывается конструктор по умолчанию без аргументов `??0c@@QAE@XZ`. Для *c2* вызывается другой конструктор `??0c@@QAE@HH@Z` и передаются два числа в качестве аргументов.

А указатель на объект (*this* в терминологии Си++) передается в регистре ECX. Это называется `thiscall (?? (стр. ??))` — метод передачи указателя на объект.

В данном случае, MSVC делает это через ECX. Необходимо помнить, что это не стандартизированный метод, и другие компиляторы могут делать это иначе, например, через первый аргумент функции (как GCC).

Почему у имен функций такие странные имена? Это [name mangling](#).

В Си++, у класса, может иметься несколько методов с одинаковыми именами, но аргументами разных типов — это полиморфизм. Ну и конечно, у разных классов могут быть методы с одинаковыми именами.

*Name mangling* позволяет закодировать имя класса + имя метода + типы всех аргументов метода в одной ASCII-строке, которая затем используется как внутреннее имя функции. Это все потому что ни компоновщик<sup>24</sup>, ни загрузчик DLL **OS!** (мангленные имена могут быть среди экспортов/импортов в DLL), ничего не знают о Си++ или **OOP!**<sup>25</sup>.

Далее вызывается два раза `dump()`.

Теперь смотрим на код в конструкторах:

Листинг 3.83: MSVC

```

_this$ = -4      ; size = 4
??0c@@QAE@XZ PROC ; c::c, COMDAT
; _this$ = ecx
  push ebp
  mov  ebp, esp
  push ecx
  mov  DWORD PTR _this$[ebp], ecx
  mov  eax, DWORD PTR _this$[ebp]
  mov  DWORD PTR [eax], 667
  mov  ecx, DWORD PTR _this$[ebp]
  mov  DWORD PTR [ecx+4], 999
  mov  eax, DWORD PTR _this$[ebp]
  mov  esp, ebp
  pop  ebp
  ret  0
??0c@@QAE@XZ ENDP ; c::c

_this$ = -4 ; size = 4
_a$ = 8 ; size = 4
_b$ = 12 ; size = 4
??0c@@QAE@HH@Z PROC ; c::c, COMDAT
; _this$ = ecx
  push ebp
  mov  ebp, esp

```

<sup>24</sup>linker

<sup>25</sup>**OOP!**

```

push ecx
mov  DWORD PTR _this$[ebp], ecx
mov  eax, DWORD PTR _this$[ebp]
mov  ecx, DWORD PTR _a$[ebp]
mov  DWORD PTR [eax], ecx
mov  edx, DWORD PTR _this$[ebp]
mov  eax, DWORD PTR _b$[ebp]
mov  DWORD PTR [edx+4], eax
mov  eax, DWORD PTR _this$[ebp]
mov  esp, ebp
pop  ebp
ret  8
??0c@@QAE@HH@Z ENDP ; c::c

```

Конструкторы — это просто функции, они используют указатель на структуру в ECX, копируют его себе в локальную переменную, хотя это и не обязательно.

Из стандарта Си++ мы знаем (C++11 12.1) что конструкторы не должны возвращать значение. В реальности, внутри, конструкторы возвращают указатель на созданный объект, т.е., *this*.

И еще метод `dump()`:

Листинг 3.84: MSVC

```

_this$ = -4 ; size = 4
?dump@@@QAEXXZ PROC ; c::dump, COMDAT
; _this$ = ecx
push ebp
mov  ebp, esp
push ecx
mov  DWORD PTR _this$[ebp], ecx
mov  eax, DWORD PTR _this$[ebp]
mov  ecx, DWORD PTR [eax+4]
push ecx
mov  edx, DWORD PTR _this$[ebp]
mov  eax, DWORD PTR [edx]
push eax
push OFFSET ??_C@_07NJBDCIEC@?.$CFd?$DL?5?$CFd?6?$AA@
call _printf
add  esp, 12
mov  esp, ebp
pop  ebp
ret  0
?dump@@@QAEXXZ ENDP ; c::dump

```

Все очень просто, `dump()` берет указатель на структуру состоящую из двух *int* через ECX, выдергивает оттуда две переменные и передает их в `printf()`.

А если скомпилировать с оптимизацией (`/Ox`), то кода будет намного меньше:

Листинг 3.85: MSVC

```

??0c@@QAE@XZ PROC ; c::c, COMDAT
; _this$ = ecx

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

    mov     eax, ecx
    mov     DWORD PTR [eax], 667
    mov     DWORD PTR [eax+4], 999
    ret     0
??0c@@QAE@XZ ENDP ; c::c

_a$ = 8 ; size = 4
_b$ = 12 ; size = 4
??0c@@QAE@HH@Z PROC ; c::c, COMDAT
; _this$ = ecx
    mov     edx, DWORD PTR _b$[esp-4]
    mov     eax, ecx
    mov     ecx, DWORD PTR _a$[esp-4]
    mov     DWORD PTR [eax], ecx
    mov     DWORD PTR [eax+4], edx
    ret     8
??0c@@QAE@HH@Z ENDP ; c::c

?dump@c@@QAE@XXZ PROC ; c::dump, COMDAT
; _this$ = ecx
    mov     eax, DWORD PTR [ecx+4]
    mov     ecx, DWORD PTR [ecx]
    push    eax
    push    ecx
    push    OFFSET ??_C@_07NJBDCIEC@?%CFd?$DL?5?$CFd?6?$AA@
    call   _printf
    add     esp, 12
    ret     0
?dump@c@@QAE@XXZ ENDP ; c::dump

```

Вот и все. Единственное о чем еще нужно сказать, это о том, что в функции `main()`, когда вызывался второй конструктор с двумя аргументами, за ним не корректировался стек при помощи `add esp, X`. В то же время, в конце конструктора вместо `RET` имеется `RET 8`.

Это потому что здесь используется `thiscall` (`??` (стр. ??)), который, вместе с `stdcall` (`??` (стр. ??)) (все это — методы передачи аргументов через стек), предлагает вызываемой функции корректировать стек. Инструкция `ret X` сначала прибавляет `X` к `ESP`, затем передает управление вызывающей функции.

См. также в соответствующем разделе о способах передачи аргументов через стек (`??` (стр. ??)).

Еще, кстати, нужно отметить, что именно компилятор решает, когда вызывать конструктор и деструктор — но это и так известно из основ языка `C++`.

## MSVC: x86-64

Как мы уже знаем, в `x86-64` первые 4 аргумента функции передаются через регистры `RCX`, `RDX`, `R8`, `R9`, а остальные — через стек. Тем не менее, указатель на объект `this` передается через `RCX`, а первый аргумент метода — в `RDX`, и т. д. Здесь это видно во внутренних методах `c(int a, int b)`:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Листинг 3.86: Оптимизирующий MSVC 2012 x64

```

; void dump()

?dump@@@QEAAAXZ PROC ; c::dump
    mov     r8d, DWORD PTR [rcx+4]
    mov     edx, DWORD PTR [rcx]
    lea     rcx, OFFSET FLAT:??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@ ;
    '%d; %d'
    jmp     printf
?dump@@@QEAAAXZ ENDP ; c::dump

; c(int a, int b)

??0c@@QEAA@HH@Z PROC ; c::c
    mov     DWORD PTR [rcx], edx ; первый аргумент: a
    mov     DWORD PTR [rcx+4], r8d ; второй аргумент: b
    mov     rax, rcx
    ret     0
??0c@@QEAA@HH@Z ENDP ; c::c

; конструктор по умолчанию

??0c@@QEAA@XZ PROC ; c::c
    mov     DWORD PTR [rcx], 667
    mov     DWORD PTR [rcx+4], 999
    mov     rax, rcx
    ret     0
??0c@@QEAA@XZ ENDP ; c::c

```

Тип *int* в x64 остается 32-битным <sup>26</sup>, поэтому здесь используются 32-битные части регистров.

В методе `dump()` вместо `RET` мы видим `JMP printf`, этот хак мы рассматривали ранее: `??` (стр. `??`).

### GCC: x86

В GCC 4.4.1 всё почти так же, за исключением некоторых различий.

Листинг 3.87: GCC 4.4.1

```

    public main
main proc near

var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
var_18 = dword ptr -18h
var_10 = dword ptr -10h
var_8  = dword ptr -8

    push ebp

```

<sup>26</sup>Видимо, так решили для упрощения портирования Си/Си++-кода на x64

```

mov  ebp, esp
and  esp, 0FFFFFF0h
sub  esp, 20h
lea  eax, [esp+20h+var_8]
mov  [esp+20h+var_20], eax
call _ZN1cC1Ev
mov  [esp+20h+var_18], 6
mov  [esp+20h+var_1C], 5
lea  eax, [esp+20h+var_10]
mov  [esp+20h+var_20], eax
call _ZN1cC1Eii
lea  eax, [esp+20h+var_8]
mov  [esp+20h+var_20], eax
call _ZN1c4dumpEv
lea  eax, [esp+20h+var_10]
mov  [esp+20h+var_20], eax
call _ZN1c4dumpEv
mov  eax, 0
leave
retn
main endp

```

Здесь мы видим, что применяется иной *name mangling* характерный для стандартов GNU <sup>27</sup> Во-вторых, указатель на экземпляр передается как первый аргумент функции — конечно же, скрыто от программиста.

Это первый конструктор:

```

_ZN1cC1Ev      public _ZN1cC1Ev ; weak
_ZN1cC1Ev      proc near          ; CODE XREF: main+10
arg_0          = dword ptr 8
               push    ebp
               mov     ebp, esp
               mov     eax, [ebp+arg_0]
               mov     dword ptr [eax], 667
               mov     eax, [ebp+arg_0]
               mov     dword ptr [eax+4], 999
               pop     ebp
               retn
_ZN1cC1Ev      endp

```

Он просто записывает два числа по указателю, переданному в первом (и единственном) аргументе.

Второй конструктор:

```

_ZN1cC1Eii     public _ZN1cC1Eii
_ZN1cC1Eii     proc near

```

<sup>27</sup>Еще о *name mangling* разных компиляторов: [Agner Fog, *Calling conventions* (2015)].

```

arg_0      = dword ptr  8
arg_4      = dword ptr  0Ch
arg_8      = dword ptr  10h

        push    ebp
        mov     ebp, esp
        mov     eax, [ebp+arg_0]
        mov     edx, [ebp+arg_4]
        mov     [eax], edx
        mov     eax, [ebp+arg_0]
        mov     edx, [ebp+arg_8]
        mov     [eax+4], edx
        pop     ebp
        retn
_ZN1cC1Eii endp

```

Это функция, аналог которой мог бы выглядеть так:

```

void ZN1cC1Eii (int *obj, int a, int b)
{
    *obj=a;
    *(obj+1)=b;
};

```

...что, в общем, предсказуемо.

И функция dump():

```

_ZN1c4dumpEv public _ZN1c4dumpEv
              proc near

var_18      = dword ptr -18h
var_14      = dword ptr -14h
var_10      = dword ptr -10h
arg_0       = dword ptr  8

        push    ebp
        mov     ebp, esp
        sub     esp, 18h
        mov     eax, [ebp+arg_0]
        mov     edx, [eax+4]
        mov     eax, [ebp+arg_0]
        mov     eax, [eax]
        mov     [esp+18h+var_10], edx
        mov     [esp+18h+var_14], eax
        mov     [esp+18h+var_18], offset aDD ; "%d; %d\n"
        call   _printf
        leave
        retn
_ZN1c4dumpEv endp

```

Эта функция *во внутреннем представлении* имеет один аргумент, через кото-

рый передается указатель на объект<sup>28</sup> (*this*).

Это можно переписать на Си:

```
void ZN1c4dumpEv (int *obj)
{
    printf ("%d; %d\n", *obj, *(obj+1));
};
```

Таким образом, если брать в учет только эти простые примеры, разница между MSVC и GCC в способе кодирования имен функций (*name mangling*) и передаче указателя на экземпляр класса (через ECX или через первый аргумент).

### GCC: x86-64

Первые 6 аргументов, как мы уже знаем, передаются через 6 регистров RDI, RSI, RDX, RCX, R8 и R9 ([Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)]<sup>29</sup>), а указатель на *this* через первый (RDI) что мы здесь и видим. Тип *int* 32-битный и здесь. Хак с JMP вместо RET используется и здесь.

Листинг 3.88: GCC 4.4.6 x64

```
; конструктор по умолчанию
_ZN1cC2Ev:
    mov     DWORD PTR [rdi], 667
    mov     DWORD PTR [rdi+4], 999
    ret

; c(int a, int b)
_ZN1cC2Eii:
    mov     DWORD PTR [rdi], esi
    mov     DWORD PTR [rdi+4], edx
    ret

; dump()
_ZN1c4dumpEv:
    mov     edx, DWORD PTR [rdi+4]
    mov     esi, DWORD PTR [rdi]
    xor     eax, eax
    mov     edi, OFFSET FLAT:.LC0 ; "%d; %d\n"
    jmp     printf
```

<sup>28</sup>экземпляр класса

<sup>29</sup>Также доступно здесь: <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>

## Наследование классов

О наследованных классах можно сказать, что это та же простая структура, которую мы уже рассмотрели, только расширяемая в наследуемых классах.

Возьмем очень простой пример:

```
#include <stdio.h>

class object
{
    public:
        int color;
        object() { };
        object (int color) { this->color=color; };
        void print_color() { printf ("color=%d\n", color); };
};

class box : public object
{
    private:
        int width, height, depth;
    public:
        box(int color, int width, int height, int depth)
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is a box. color=%d, width=%d, height=%d, depth=%d\n", color, width, height, depth);
        };
};

class sphere : public object
{
    private:
        int radius;
    public:
        sphere(int color, int radius)
        {
            this->color=color;
            this->radius=radius;
        };
        void dump()
        {
            printf ("this is sphere. color=%d, radius=%d\n", color, radius);
        };
};

int main()
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

{
    box b(1, 10, 20, 30);
    sphere s(2, 40);

    b.print_color();
    s.print_color();

    b.dump();
    s.dump();

    return 0;
};

```

Исследуя сгенерированный код для функций/методов `dump()`, а также `object::print_color()`, посмотрим, какая будет разметка памяти для структур-объектов (для 32-битного кода).

Итак, методы `dump()` разных классов сгенерированные MSVC 2008 с опциями `/Ox` и `/Ob0`<sup>30</sup>.

Листинг 3.89: Оптимизирующий MSVC 2008 /Ob0

```

??_C@_09GCEd0LPA@color?$DN?$CFd?6?$AA@ DB 'color=%d', 0aH, 00H ; `string'
?print_color@object@@QAEXXZ PROC ; object::print_color, COMDAT
; _this$ = ecx
    mov     eax, DWORD PTR [ecx]
    push   eax

; 'color=%d', 0aH, 00H
    push   OFFSET ??_C@_09GCEd0LPA@color?$DN?$CFd?6?$AA@
    call   _printf
    add    esp, 8
    ret    0
?print_color@object@@QAEXXZ ENDP ; object::print_color

```

Листинг 3.90: Оптимизирующий MSVC 2008 /Ob0

```

?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
    mov     eax, DWORD PTR [ecx+12]
    mov     edx, DWORD PTR [ecx+8]
    push   eax
    mov     eax, DWORD PTR [ecx+4]
    mov     ecx, DWORD PTR [ecx]
    push   edx
    push   eax
    push   ecx

;
    'this is a box. color=%d, width=%d, height=%d, depth=%d', 0aH, 00H ; `string'
    push   OFFSET ??_C@_0Dg@NCNGAADL@this?5is?5box?4?5color?$DN?$CFd?0?5width?
    ↵ ?$DN?$CFd?0@

```

<sup>30</sup> опция `/Ob0` означает отмену `inline expansion`, ведь вставка компилятором тела функции/метода прямо в код где он вызывается, может затруднить наши эксперименты.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

call _printf
add esp, 20
ret 0
?dump@box@@QAEXXZ ENDP ; box::dump

```

Листинг 3.91: Оптимизирующий MSVC 2008 /Ob0

```

?dump@sphere@@QAEXXZ PROC ; sphere::dump, COMDAT
; _this$ = ecx
mov eax, DWORD PTR [ecx+4]
mov ecx, DWORD PTR [ecx]
push eax
push ecx

; 'this is sphere. color=%d, radius=%d', 0aH, 00H
push OFFSET ??_C@_0CF@EFEDJLDC@this?5is?5sphere?4?5color?$DN?$CFd?0?5↵
↳ radius@
call _printf
add esp, 12
ret 0
?dump@sphere@@QAEXXZ ENDP ; sphere::dump

```

Итак, разметка полей получается следующая:

(базовый класс *object*)

смещение	описание
+0x0	int color

(унаследованные классы)

*box*:

смещение	описание
+0x0	int color
+0x4	int width
+0x8	int height
+0xC	int depth

*sphere*:

смещение	описание
+0x0	int color
+0x4	int radius

Посмотрим тело `main()`:

Листинг 3.92: Оптимизирующий MSVC 2008 /Ob0

```

PUBLIC _main
_TEXT SEGMENT
_s$ = -24 ; size = 8
_b$ = -16 ; size = 16
_main PROC
sub esp, 24
push 30

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

push 20
push 10
push 1
lea ecx, DWORD PTR _b$[esp+40]
call ??0box@@QAE@HHH@Z ; box::box
push 40
push 2
lea ecx, DWORD PTR _s$[esp+32]
call ??0sphere@@QAE@HH@Z ; sphere::sphere
lea ecx, DWORD PTR _b$[esp+24]
call ?print_color@object@@QAEHXZ ; object::print_color
lea ecx, DWORD PTR _s$[esp+24]
call ?print_color@object@@QAEHXZ ; object::print_color
lea ecx, DWORD PTR _b$[esp+24]
call ?dump@box@@QAEHXZ ; box::dump
lea ecx, DWORD PTR _s$[esp+24]
call ?dump@sphere@@QAEHXZ ; sphere::dump
xor eax, eax
add esp, 24
ret 0
_main ENDP

```

Наследованные классы всегда должны добавлять свои поля после полей базового класса для того, чтобы методы базового класса могли продолжать работать со своими собственными полями.

Когда метод `object::print_color()` вызывается, ему в качестве `this` передается указатель и на объект типа `box` и на объект типа `sphere`, так как он может легко работать с классами `box` и `sphere`, потому что поле `color` в этих классах всегда стоит по тому же адресу (по смещению `0x0`).

Можно также сказать, что методу `object::print_color()` даже не нужно знать, с каким классом он работает, до тех пор, пока будет соблюдаться условие *закрепления* полей по тем же адресам, а это условие соблюдается всегда.

А если вы создадите класс-наследник класса `box`, например, то компилятор будет добавлять новые поля уже за полем `depth`, оставляя уже имеющиеся поля класса `box` по тем же адресам.

Так, метод `box::dump()` будет нормально работать обращаясь к полям `color`, `width`, `height` и `depth`, всегда находящимся по известным адресам.

Код на GCC практически точно такой же, за исключением способа передачи `this` (он, как уже было указано, передается в первом аргументе, вместо регистра ECX).

## Инкапсуляция

Инкапсуляция — это сокрытие данных в *private* секциях класса, например, чтобы разрешить доступ к ним только для методов этого класса, но не более.

Однако, маркируется ли как-нибудь в коде тот сам факт, что некоторое поле — приватное, а некоторое другое — нет?

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Нет, никак не маркируется.

Попробуем простой пример:

```
#include <stdio.h>

class box
{
private:
    int color, width, height, depth;
public:
    box(int color, int width, int height, int depth)
    {
        this->color=color;
        this->width=width;
        this->height=height;
        this->depth=depth;
    };
    void dump()
    {
        printf ("this is a box. color=%d, width=%d, height=%d, depth=%d\n",
        color, width, height, depth);
    };
};
```

Снова скомпилируем в MSVC 2008 с опциями /Ox и /Ob0 и посмотрим код метода box::dump():

```
?dump@box@@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
    mov     eax, DWORD PTR [ecx+12]
    mov     edx, DWORD PTR [ecx+8]
    push   eax
    mov     eax, DWORD PTR [ecx+4]
    mov     ecx, DWORD PTR [ecx]
    push   edx
    push   eax
    push   ecx
; 'this is a box. color=%d, width=%d, height=%d, depth=%d', 0aH, 00H
    push   OFFSET ??_C@_0DG@NCNGAADL@this?5is?5box?4?5color?5DN?5$CFd?0?5width?
    call   _printf
    add     esp, 20
    ret     0
?dump@box@@@QAEXXZ ENDP ; box::dump
```

Разметка полей в классе выходит такой:

смещение	описание
+0x0	int color
+0x4	int width
+0x8	int height
+0xC	int depth

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Все поля приватные и недоступные для модификации из других функций, но, зная эту разметку, сможем ли мы создать код модифицирующий эти поля?

Для этого добавим функцию `hack_oop_encapsulation()`, которая если обладает приведенным ниже телом, то просто не скомпилируется:

```
void hack_oop_encapsulation(class box * o)
{
    o->width=1; // этот код не может быть скомпилирован:
                // "error C2248: 'box::width' : cannot access private member
                // declared in class 'box'"
};
```

Тем не менее, если преобразовать тип `box` к типу *указатель на массив int*, и если модифицировать полученный массив *int*-ов, тогда всё получится.

```
void hack_oop_encapsulation(class box * o)
{
    unsigned int *ptr_to_object=reinterpret_cast<unsigned int*>(o);
    ptr_to_object[1]=123;
};
```

Код этой функции довольно прост — можно сказать, функция берет на вход указатель на массив *int*-ов и записывает `123` во второй *int*:

```
?hack_oop_encapsulation@@YAXPAVbox@@@Z PROC ; hack_oop_encapsulation
    mov eax, DWORD PTR _o$[esp-4]
    mov DWORD PTR [eax+4], 123
    ret 0
?hack_oop_encapsulation@@YAXPAVbox@@@Z ENDP ; hack_oop_encapsulation
```

Проверим, как это работает:

```
int main()
{
    box b(1, 10, 20, 30);

    b.dump();

    hack_oop_encapsulation(&b);

    b.dump();

    return 0;
};
```

Запускаем:

```
this is a box. color=1, width=10, height=20, depth=30
this is a box. color=1, width=123, height=20, depth=30
```

Выходит, инкапсуляция — это защита полей класса только на стадии компиляции.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Компилятор ЯП Си++ не позволяет сгенерировать код прямо модифицирующий защищенные поля, тем не менее, используя *грязные трюки* — это вполне возможно.

### Множественное наследование

Множественное наследование — это создание класса наследующего поля и методы от двух или более классов.

Снова напишем простой пример:

```
#include <stdio.h>

class box
{
public:
    int width, height, depth;
    box() { };
    box(int width, int height, int depth)
    {
        this->width=width;
        this->height=height;
        this->depth=depth;
    };
    void dump()
    {
        printf ("this is a box. width=%d, height=%d, depth=%d\n", width, height, depth);
    };
    int get_volume()
    {
        return width * height * depth;
    };
};

class solid_object
{
public:
    int density;
    solid_object() { };
    solid_object(int density)
    {
        this->density=density;
    };
    int get_density()
    {
        return density;
    };
    void dump()
    {
        printf ("this is a solid_object. density=%d\n", density);
    };
};
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

class solid_box: box, solid_object
{
public:
    solid_box (int width, int height, int depth, int density)
    {
        this->width=width;
        this->height=height;
        this->depth=depth;
        this->density=density;
    };
    void dump()
    {
        printf ("this is a solid_box. width=%d, height=%d, depth=%d, ↵
↳ density=%d\n", width, height, depth, density);
    };
    int get_weight() { return get_volume() * get_density(); };
};

int main()
{
    box b(10, 20, 30);
    solid_object so(100);
    solid_box sb(10, 20, 30, 3);

    b.dump();
    so.dump();
    sb.dump();
    printf ("%d\n", sb.get_weight());

    return 0;
};

```

Снова скомпилируем в MSVC 2008 с опциями /Ox и /Ob0 и посмотрим код методов box::dump(), solid\_object::dump() и solid\_box::dump():

Листинг 3.93: Оптимизирующий MSVC 2008 /Ob0

```

?dump@box@@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
    mov     eax, DWORD PTR [ecx+8]
    mov     edx, DWORD PTR [ecx+4]
    push   eax
    mov     eax, DWORD PTR [ecx]
    push   edx
    push   eax
; 'this is a box. width=%d, height=%d, depth=%d', 0aH, 00H
    push   OFFSET ??_C@_0CM@DIKPHDFI@this?5is?5box?4?5width?5DN?5CFd?0?5↳
↳ height?5DN?5CFd@
    call   _printf
    add     esp, 16
    ret     0
?dump@box@@@QAEXXZ ENDP ; box::dump

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Листинг 3.94: Оптимизирующий MSVC 2008 /Ob0

```
?dump@solid_object@@QAEXXZ PROC ; solid_object::dump, COMDAT
; _this$ = ecx
  mov  eax, DWORD PTR [ecx]
  push eax
; 'this is a solid_object. density=%d', 0aH
  push OFFSET ??_C@_0CC@KICFJINL@this?5is?5solid_object?4?5density?$DN?
  ↵ $CFd@
  call _printf
  add  esp, 8
  ret  0
?dump@solid_object@@QAEXXZ ENDP ; solid_object::dump
```

Листинг 3.95: Оптимизирующий MSVC 2008 /Ob0

```
?dump@solid_box@@QAEXXZ PROC ; solid_box::dump, COMDAT
; _this$ = ecx
  mov  eax, DWORD PTR [ecx+12]
  mov  edx, DWORD PTR [ecx+8]
  push eax
  mov  eax, DWORD PTR [ecx+4]
  mov  ecx, DWORD PTR [ecx]
  push edx
  push eax
  push ecx
; 'this is a solid_box. width=%d, height=%d, depth=%d, density=%d', 0aH
  push OFFSET ??_C@_0D0@HNCNIHNN@this?5is?5solid_box?4?5width?$DN?$CFd
  ↵ ?0?5hei@
  call _printf
  add  esp, 20
  ret  0
?dump@solid_box@@QAEXXZ ENDP ; solid_box::dump
```

Выходит, имеем такую разметку в памяти для всех трех классов:

класс *box*:

смещение	описание
+0x0	width
+0x4	height
+0x8	depth

класс *solid\_object*:

смещение	описание
+0x0	density

Можно сказать, что разметка класса *solid\_box* объединённая:

Класс *solid\_box*:

смещение	описание
+0x0	width
+0x4	height
+0x8	depth
+0xC	density

Код методов `box::get_volume()` и `solid_object::get_density()` тривиален:

Листинг 3.96: Оптимизирующий MSVC 2008 /Ob0

```
?get_volume@box@@QAEHXZ PROC ; box::get_volume, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+8]
    imul eax, DWORD PTR [ecx+4]
    imul eax, DWORD PTR [ecx]
    ret  0
?get_volume@box@@QAEHXZ ENDP ; box::get_volume
```

Листинг 3.97: Оптимизирующий MSVC 2008 /Ob0

```
?get_density@solid_object@@QAEHXZ PROC ; solid_object::get_density, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx]
    ret  0
?get_density@solid_object@@QAEHXZ ENDP ; solid_object::get_density
```

А вот код метода `solid_box::get_weight()` куда интереснее:

Листинг 3.98: Оптимизирующий MSVC 2008 /Ob0

```
?get_weight@solid_box@@QAEHXZ PROC ; solid_box::get_weight, COMDAT
; _this$ = ecx
    push esi
    mov  esi, ecx
    push edi
    lea  ecx, DWORD PTR [esi+12]
    call ?get_density@solid_object@@QAEHXZ ; solid_object::get_density
    mov  ecx, esi
    mov  edi, eax
    call ?get_volume@box@@QAEHXZ ; box::get_volume
    imul eax, edi
    pop  edi
    pop  esi
    ret  0
?get_weight@solid_box@@QAEHXZ ENDP ; solid_box::get_weight
```

`get_weight()` просто вызывает два метода, но для `get_volume()` он передает просто указатель на `this`, а для `get_density()`, он передает указатель на `this` сдвинутый на 12 байт (либо 0xC байт), а там, в разметке класса `solid_box`, как раз начинаются поля класса `solid_object`.

Так, метод `solid_object::get_density()` будет полагать что работает с обычным классом

`solid_object`, а метод `box::get_volume()` будет работать только со своими тремя полями, полагая, что работает с обычным экземпляром класса `box`.

Таким образом, можно сказать, что экземпляр класса-наследника нескольких классов представляет в памяти просто *объединённый* класс, содержащий все унаследованные поля. А каждый унаследованный метод вызывается с передачей ему указателя на соответствующую часть структуры.

## Виртуальные методы

И снова простой пример:

```
#include <stdio.h>

class object
{
    public:
        int color;
        object() { };
        object (int color) { this->color=color; };
        virtual void dump()
        {
            printf ("color=%d\n", color);
        };
};

class box : public object
{
    private:
        int width, height, depth;
    public:
        box(int color, int width, int height, int depth)
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is a box. color=%d, width=%d, height=%d, depth=%d\n",
                color, width, height, depth);
        };
};

class sphere : public object
{
    private:
        int radius;
    public:
        sphere(int color, int radius)
        {
            this->color=color;
        };
};
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

        this->radius=radius;
    };
    void dump()
    {
        printf ("this is sphere. color=%d, radius=%d\n", color, radius);
    };
};

int main()
{
    box b(1, 10, 20, 30);
    sphere s(2, 40);

    object *o1=&b;
    object *o2=&s;

    o1->dump();
    o2->dump();
    return 0;
};

```

У класса *object* есть виртуальный метод `dump()`, впоследствии заменяемый в классах-наследниках *box* и *sphere*.

Если в какой-то среде, где неизвестно, какого типа является экземпляр класса, как в функции `main()` в примере, вызывается виртуальный метод `dump()`, где-то должна сохраняться информация о том, какой же метод в итоге вызвать.

Скомпилируем в MSVC 2008 с опциями `/Ox` и `/Ob0` и посмотрим код функции `main()`:

```

_s$ = -32 ; size = 12
_b$ = -20 ; size = 20
_main PROC
    sub esp, 32
    push 30
    push 20
    push 10
    push 1
    lea ecx, DWORD PTR _b$[esp+48]
    call ??0box@@QAE@HHH@Z ; box::box
    push 40
    push 2
    lea ecx, DWORD PTR _s$[esp+40]
    call ??0sphere@@QAE@HH@Z ; sphere::sphere
    mov eax, DWORD PTR _b$[esp+32]
    mov edx, DWORD PTR [eax]
    lea ecx, DWORD PTR _b$[esp+32]
    call edx
    mov eax, DWORD PTR _s$[esp+32]
    mov edx, DWORD PTR [eax]
    lea ecx, DWORD PTR _s$[esp+32]
    call edx

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

xor  eax, eax
add  esp, 32
ret  0
_main ENDP

```

Указатель на функцию `dump()` берется откуда-то из экземпляра класса (объекта). Где мог записаться туда адрес нового метода-функции? Только в конструкторах, больше нигде: ведь в функции `main()` ничего более не вызывается.

31

Посмотрим код конструктора класса `box`:

```

??_R0?AVbox@@@8 DD FLAT:??_7type_info@@6B@ ; box `RTTI Type Descriptor'
  DD  00H
  DB  '?.AVbox@', 00H

??_R1A@?0A@EA@box@@@8 DD FLAT:??_R0?AVbox@@@8 ; box::`RTTI Base Class
Descriptor at (0,-1,0,64)'
  DD  01H
  DD  00H
  DD  0ffffffffH
  DD  00H
  DD  040H
  DD  FLAT:??_R3box@@@8

??_R2box@@@8 DD FLAT:??_R1A@?0A@EA@box@@@8 ; box::`RTTI Base Class Array'
  DD  FLAT:??_R1A@?0A@EA@object@@@8

??_R3box@@@8 DD 00H ; box::`RTTI Class Hierarchy Descriptor'
  DD  00H
  DD  02H
  DD  FLAT:??_R2box@@@8

??_R4box@@@6B@ DD 00H ; box::`RTTI Complete Object Locator'
  DD  00H
  DD  00H
  DD  FLAT:??_R0?AVbox@@@8
  DD  FLAT:??_R3box@@@8

??_7box@@@6B@ DD FLAT:??_R4box@@@6B@ ; box::`vftable'
  DD  FLAT:?dump@box@@@UAEXXZ

_color$ = 8 ; size = 4
_width$ = 12 ; size = 4
_height$ = 16 ; size = 4
_depth$ = 20 ; size = 4
??0box@@@QAE@HHHH@Z PROC ; box::box, COMDAT
; _this$ = ecx
  push esi
  mov esi, ecx
  call ??0object@@@QAE@XZ ; object::object
  mov eax, DWORD PTR _color$[esp]

```

<sup>31</sup>Об указателях на функции читайте больше в соответствующем разделе: (?? (стр. ??))

```

mov ecx, DWORD PTR _width$[esp]
mov edx, DWORD PTR _height$[esp]
mov DWORD PTR [esi+4], eax
mov eax, DWORD PTR _depth$[esp]
mov DWORD PTR [esi+16], eax
mov DWORD PTR [esi], OFFSET ??_7box@@6B@
mov DWORD PTR [esi+8], ecx
mov DWORD PTR [esi+12], edx
mov eax, esi
pop esi
ret 16
??0box@@@QAE@HHHH@Z ENDP ; box::box

```

Здесь мы видим, что разметка класса немного другая: в качестве первого поля имеется указатель на некую таблицу `box::`vftable'` (название оставлено компилятором MSVC).

В этой таблице есть ссылка на таблицу с названием `box::`RTTI Complete Object Locator'` и еще ссылка на метод `box::dump()`.

Итак, это называется таблица виртуальных методов и **RTTI!**<sup>32</sup>. Таблица виртуальных методов хранит в себе адреса методов, а **RTTI!** хранит информацию о типах вообще.

Кстати, **RTTI!**-таблицы — это именно те таблицы, информация из которых используются при вызове `dynamic_cast` и `typeid` в Си++. Вы можете увидеть, что здесь хранится даже имя класса в виде обычной строки.

Так, какой-нибудь метод базового класса `object` может вызвать виртуальный метод `object::dump()` что в итоге вызовет нужный метод унаследованного класса, потому что информация о нем присутствует прямо в этой структуре класса.

Работа с этими таблицами и поиск адреса нужного метода, занимает какое-то время процессора, возможно, поэтому считается что работа с виртуальными методами медленна.

В сгенерированном коде от GCC **RTTI!**-таблицы устроены чуть-чуть иначе.

### 3.19.2. ostream

Начнем снова с примера типа «hello world», на этот раз используя `ostream`:

```

#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}

```

Из практически любого учебника Си++, известно, что операцию `<<` можно определить (или перегрузить — *overload*) для других типов.

<sup>32</sup>**RTTI!**

Что и делается в ostream. Видно, что в реальности вызывается оператор<< для ostream:

Листинг 3.99: MSVC 2012 (reduced listing)

```
$SG37112 DB 'Hello, world!', 0aH, 00H

_main PROC
    push OFFSET $SG37112
    push OFFSET ?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ;
    std::cout
    call ???$?6U?$char_traits@D@std@@@std@@YAAAV?$basic_ostream@DU?
    ↪ $char_traits@D@std@@@0@AAV10@PBD@Z ;
    std::operator<<<std::char_traits<char> >
    add esp, 8
    xor eax, eax
    ret 0
_main ENDP
```

Немного переделаем пример:

```
#include <iostream>

int main()
{
    std::cout << "Hello, " << "world!\n";
}
```

И снова, из многих учебников по Си++, известно, что результат каждого оператор<< в ostream передается в следующий.

Действительно:

Листинг 3.100: MSVC 2012

```
$SG37112 DB 'world!', 0aH, 00H
$SG37113 DB 'Hello, ', 00H

_main PROC
    push OFFSET $SG37113 ; 'Hello, '
    push OFFSET ?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ;
    std::cout
    call ???$?6U?$char_traits@D@std@@@std@@YAAAV?$basic_ostream@DU?
    ↪ $char_traits@D@std@@@0@AAV10@PBD@Z ;
    std::operator<<<std::char_traits<char> >
    add esp, 8

    push OFFSET $SG37112 ; 'world!'
    push eax ; результат работы предыдущей ф-ции
    call ???$?6U?$char_traits@D@std@@@std@@YAAAV?$basic_ostream@DU?
    ↪ $char_traits@D@std@@@0@AAV10@PBD@Z ;
    std::operator<<<std::char_traits<char> >
    add esp, 8

    xor eax, eax
    ret 0
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
_main ENDP
```

Если переименовать название метода `operator<<` в `f()`, то этот код выглядел бы так:

```
f(f(std::cout, "Hello, "), "world!");
```

GCC генерирует практически такой же код как и MSVC.

### 3.19.3. References

References в Си++ это тоже указатели (?? (стр. ??)), но их называют *безопасными* (safe), потому что работая с ними, труднее сделать ошибку (C++11 8.3.2). Например, reference всегда должен указывать объект того же типа и не может быть NULL [Marshall Cline, C++ FAQ8.6].

Более того, reference нельзя менять, нельзя его заставить указывать на другой объект (reseat) [Marshall Cline, C++ FAQ8.5].

Если мы попробуем изменить пример с указателями (?? (стр. ??)) чтобы он использовал reference вместо указателей ...

```
void f2 (int x, int y, int & sum, int & product)
{
    sum=x+y;
    product=x*y;
};
```

...то выяснится, что скомпилированный код абсолютно такой же как и в примере с указателями (?? (стр. ??)):

Листинг 3.101: Оптимизирующий MSVC 2010

```
_x$ = 8      ; size = 4
_y$ = 12     ; size = 4
_sum$ = 16   ; size = 4
_product$ = 20 ; size = 4
?f2@@YAXHHAAN0@Z PROC ; f2
    mov     ecx, DWORD PTR _y$[esp-4]
    mov     eax, DWORD PTR _x$[esp-4]
    lea    edx, DWORD PTR [eax+ecx]
    imul   eax, ecx
    mov     ecx, DWORD PTR _product$[esp-4]
    push   esi
    mov     esi, DWORD PTR _sum$[esp]
    mov     DWORD PTR [esi], edx
    mov     DWORD PTR [ecx], eax
    pop    esi
    ret    0
?f2@@YAXHHAAN0@Z ENDP ; f2
```

(Почему у функций в Си++ такие странные имена, описано здесь: ?? (стр. ??).)

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Следовательно, references в C++ эффективны настолько, насколько и обычные указатели.

### 3.19.4. STL

N.B.: все примеры здесь были проверены только в 32-битной среде. x64-версии не были проверены.

#### std::string

##### Как устроена структура

Многие строковые библиотеки [Денис Юричев, *Заметки о языке программирования Си/Си++2.2*] обеспечивают структуру содержащую ссылку на буфер собственно со строкой, переменную всегда содержащую длину строки (что очень удобно для массы функций [Денис Юричев, *Заметки о языке программирования Си/Си++2.2.1*]) и переменную содержащую текущий размер буфера.

Строка в буфере обыкновенно оканчивается нулем: это для того чтобы указатель на буфер можно было передавать в функции требующие на вход обычную **ASCII**-строку.

Стандарт Си++ не описывает, как именно нужно реализовывать std::string, но, как правило, они реализованы как описано выше, с небольшими дополнениями.

Строки в Си++ это не класс (как, например, QString в Qt), а темплейт (basic\_string), это сделано для того чтобы поддерживать строки содержащие разного типа символы: как минимум *char* и *wchar\_t*.

Так что, std::string это класс с базовым типом *char*.

А std::wstring это класс с базовым типом *wchar\_t*.

#### MSVC

В реализации MSVC, вместо ссылки на буфер может содержаться сам буфер (если строка короче 16-и символов).

Это означает, что каждая короткая строка будет занимать в памяти по крайней мере  $16 + 4 + 4 = 24$  байт для 32-битной среды либо  $16 + 8 + 8 = 32$  байта в 64-битной, а если строка длиннее 16-и символов, то прибавьте еще длину самой строки.

Листинг 3.102: пример для MSVC

```
#include <string>
#include <stdio.h>

struct std_string
{
    union
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    {
        char buf[16];
        char* ptr;
    } u;
    size_t size; // AKA 'Mysize' в MSVC
    size_t capacity; // AKA 'Myres' в MSVC
};

void dump_std_string(std::string s)
{
    struct std_string *p=(struct std_string*)&s;
    printf ("%s] size:%d capacity:%d\n", p->size>16 ? p->u.ptr : p->u.buf, ↵
    ↵ p->size, p->capacity);
};

int main()
{
    std::string s1="a short string";
    std::string s2="a string longer than 16 bytes";

    dump_std_string(s1);
    dump_std_string(s2);

    // это работает без использования c_str()
    printf ("%s\n", &s1);
    printf ("%s\n", s2);
};

```

Собственно, из этого исходника почти всё ясно.

Несколько замечаний:

Если строка короче 16-и символов, то отдельный буфер для строки в [куче](#) выделяться не будет.

Это удобно потому что на практике, основная часть строк действительно короткие. Вероятно, разработчики в Microsoft выбрали размер в 16 символов как разумный баланс.

Теперь очень важный момент в конце функции main(): мы не пользуемся методом c\_str(), тем не менее, если это скомпилировать и запустить, то обе строки появятся в консоли!

Работает это вот почему.

В первом случае строка короче 16-и символов и в начале объекта std::string (его можно рассматривать просто как структуру) расположен буфер с этой строкой. printf() трактует указатель как указатель на массив символов оканчивающийся нулем и поэтому всё работает.

Вывод второй строки (длиннее 16-и символов) даже еще опаснее: это вообще типичная программистская ошибка (или опечатка), забыть дописать c\_str(). Это работает потому что в это время в начале структуры расположен указатель на буфер. Это может надолго остаться незамеченным: до тех пока там не появится строка короче 16-и символов, тогда процесс упадет.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

**GCC**

В реализации GCC в структуре есть еще одна переменная — reference count.

Интересно, что указатель на экземпляр класса `std::string` в GCC указывает не на начало самой структуры, а на указатель на буфера. В `libstdc++-v3\include\bits\basic_string.h` мы можем прочитать что это сделано для удобства отладки:

```
* The reason you want _M_data pointing to the character %array and
* not the _Rep is so that the debugger can see the string
* contents. (Probably we should add a non-inline member to get
* the _Rep for the debugger to use, so users can check the actual
* string length.)
```

[исходный код basic\\_string.h](#)

В нашем примере мы учитываем это:

Листинг 3.103: пример для GCC

```
#include <string>
#include <stdio.h>

struct std_string
{
    size_t length;
    size_t capacity;
    size_t refcount;
};

void dump_std_string(std::string s)
{
    char *p1=*(char**)&s; // обход проверки типов GCC
    struct std_string *p2=(struct std_string*)(p1-sizeof(struct std_string)↵
    ↵);
    printf ("%s] size:%d capacity:%d\n", p1, p2->length, p2->capacity);
};

int main()
{
    std::string s1="a short string";
    std::string s2="a string longer than 16 bytes";

    dump_std_string(s1);
    dump_std_string(s2);

    // обход проверки типов GCC:
    printf ("%s\n", *(char**)&s1);
    printf ("%s\n", *(char**)&s2);
};
```

Нужны еще небольшие хаки чтобы симитировать типичную ошибку, которую мы уже видели выше, из-за более ужесточенной проверки типов в GCC, тем не менее, `printf()` работает и здесь без `c_str()`.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



**Чуть более сложный пример**

```
#include <string>
#include <stdio.h>

int main()
{
    std::string s1="Hello, ";
    std::string s2="world!\n";
    std::string s3=s1+s2;

    printf ("%s\n", s3.c_str());
}
```

Листинг 3.104: MSVC 2012

```
$SG39512 DB 'Hello, ', 00H
$SG39514 DB 'world!', 0aH, 00H
$SG39581 DB '%s', 0aH, 00H

_s2$ = -72 ; size = 24
_s3$ = -48 ; size = 24
_s1$ = -24 ; size = 24
_main PROC
    sub     esp, 72

    push   7
    push   OFFSET $SG39512
    lea   ecx, DWORD PTR _s1$[esp+80]
    mov   DWORD PTR _s1$[esp+100], 15
    mov   DWORD PTR _s1$[esp+96], 0
    mov   BYTE PTR _s1$[esp+80], 0
    call  ?assign@?$basic_string@DU?$char_traits@D@std@@V?
    ↪ $allocator@D@2@@std@@QAEAAV12@PBIDI@Z ;
    std::basic_string<char,std::char_traits<char>,std::allocator<char> >::assign

    push   7
    push   OFFSET $SG39514
    lea   ecx, DWORD PTR _s2$[esp+80]
    mov   DWORD PTR _s2$[esp+100], 15
    mov   DWORD PTR _s2$[esp+96], 0
    mov   BYTE PTR _s2$[esp+80], 0
    call  ?assign@?$basic_string@DU?$char_traits@D@std@@V?
    ↪ $allocator@D@2@@std@@QAEAAV12@PBIDI@Z ;
    std::basic_string<char,std::char_traits<char>,std::allocator<char> >::assign

    lea   eax, DWORD PTR _s2$[esp+72]
    push  eax
    lea   eax, DWORD PTR _s1$[esp+76]
    push  eax
    lea   eax, DWORD PTR _s3$[esp+80]
    push  eax
    call  ???H?DU?$char_traits@D@std@@V?$allocator@D@1@std@@YA?AV?
    ↪ $basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@@ABV10@@@Z ;
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

std::operator+<char,std::char_traits<char>,std::allocator<char> >

; вставленный код метода (inlined) c_str():
cmp  DWORD PTR _s3$[esp+104], 16
lea  eax, DWORD PTR _s3$[esp+84]
cmovae eax, DWORD PTR _s3$[esp+84]

push eax
push OFFSET $SG39581
call _printf
add  esp, 20

cmp  DWORD PTR _s3$[esp+92], 16
jb   SHORT $LN119@main
push DWORD PTR _s3$[esp+72]
call ???@YAXPAX@Z           ; operator delete
add  esp, 4
$LN119@main:
cmp  DWORD PTR _s2$[esp+92], 16
mov  DWORD PTR _s3$[esp+92], 15
mov  DWORD PTR _s3$[esp+88], 0
mov  BYTE PTR _s3$[esp+72], 0
jb   SHORT $LN151@main
push DWORD PTR _s2$[esp+72]
call ???@YAXPAX@Z           ; operator delete
add  esp, 4
$LN151@main:
cmp  DWORD PTR _s1$[esp+92], 16
mov  DWORD PTR _s2$[esp+92], 15
mov  DWORD PTR _s2$[esp+88], 0
mov  BYTE PTR _s2$[esp+72], 0
jb   SHORT $LN195@main
push DWORD PTR _s1$[esp+72]
call ???@YAXPAX@Z           ; operator delete
add  esp, 4
$LN195@main:
xor  eax, eax
add  esp, 72
ret  0
_main ENDP

```

Собственно, компилятор не конструирует строки статически: да в общем-то и как это возможно, если буфер с ней нужно хранить в [куче](#)?

Вместо этого в сегменте данных хранятся обычные **ASCIIZ!**-строки, а позже, во время выполнения, при помощи метода «assign», конструируются строки s1 и s2 . При помощи operator+, создается строка s3.

Обратите внимание на то что вызов метода c\_str() отсутствует, потому что его код достаточно короткий и компилятор вставил его прямо здесь: если строка короче 16-и байт, то в регистре EAX остается указатель на буфер, а если длиннее, то из этого же места достается адрес на буфер расположенный в [куче](#).

Далее следуют вызовы трех деструкторов, причем, они вызываются только ес-

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

ли строка длиннее 16-и байт: тогда нужно освободить буфера в [куче](#). В противном случае, так как все три объекта `std::string` хранятся в стеке, они освобождаются автоматически после выхода из функции.

Следовательно, работа с короткими строками более быстрая из-за меньшего обращения к [куче](#).

Код на GCC даже проще (из-за того, что в GCC, как мы уже видели, не реализована возможность хранить короткую строку прямо в структуре):

Листинг 3.105: GCC 4.8.1

```
.LC0:
.string "Hello, "
.LC1:
.string "world!\n"
main:
    push ebp
    mov  ebp, esp
    push edi
    push esi
    push ebx
    and  esp, -16
    sub  esp, 32
    lea  ebx, [esp+28]
    lea  edi, [esp+20]
    mov  DWORD PTR [esp+8], ebx
    lea  esi, [esp+24]
    mov  DWORD PTR [esp+4], OFFSET FLAT:.LC0
    mov  DWORD PTR [esp], edi

    call _ZNSsC1EPKcRKSaIcE

    mov  DWORD PTR [esp+8], ebx
    mov  DWORD PTR [esp+4], OFFSET FLAT:.LC1
    mov  DWORD PTR [esp], esi

    call _ZNSsC1EPKcRKSaIcE

    mov  DWORD PTR [esp+4], edi
    mov  DWORD PTR [esp], ebx

    call _ZNSsC1ERKSs

    mov  DWORD PTR [esp+4], esi
    mov  DWORD PTR [esp], ebx

    call _ZNSs6appendERKSs

    ; вставленный код метода (inlined) c_str():
    mov  eax, DWORD PTR [esp+28]
    mov  DWORD PTR [esp], eax

    call puts
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov  eax, DWORD PTR [esp+28]
lea  ebx, [esp+19]
mov  DWORD PTR [esp+4], ebx
sub  eax, 12
mov  DWORD PTR [esp], eax
call _ZNSs4_Rep10_M_disposeERKSaIcE
mov  eax, DWORD PTR [esp+24]
mov  DWORD PTR [esp+4], ebx
sub  eax, 12
mov  DWORD PTR [esp], eax
call _ZNSs4_Rep10_M_disposeERKSaIcE
mov  eax, DWORD PTR [esp+20]
mov  DWORD PTR [esp+4], ebx
sub  eax, 12
mov  DWORD PTR [esp], eax
call _ZNSs4_Rep10_M_disposeERKSaIcE
lea  esp, [ebp-12]
xor  eax, eax
pop  ebx
pop  esi
pop  edi
pop  ebp
ret

```

Можно заметить, что в деструкторы передается не указатель на объект, а указатель на место за 12 байт (или 3 слова) перед ним, то есть, на настоящее начало структуры.

### **std::string как глобальная переменная**

Опытные программисты на Си++ знают, что глобальные переменные **STL!**<sup>33</sup> типов вполне можно объявлять.

Да, действительно:

```

#include <stdio.h>
#include <string>

std::string s="a string";

int main()
{
    printf ("%s\n", s.c_str());
};

```

Но как и где будет вызываться конструктор `std::string`?

На самом деле, эта переменная будет инициализирована даже перед началом `main()`.

<sup>33</sup>**STL!**

Листинг 3.106: MSVC 2012: здесь конструируется глобальная переменная, а также регистрируется её деструктор

```

??_Es@YAXXZ PROC
    push 8
    push OFFSET $SG39512 ; 'a string'
    mov ecx, OFFSET ?s@3V?$basic_string@DU?$char_traits@D@std@@V?↵
    ↵ $allocator@D@2@std@@A ;
    S
    call ?assign@?$basic_string@DU?$char_traits@D@std@@V?↵
    ↵ $allocator@D@2@std@@QAEAAV12@PBDI@Z ;
    std::basic_string<char,std::char_traits<char>,std::allocator<char> >::assign
    push OFFSET ??_Fs@YAXXZ ; `dynamic atexit destructor for 's'`
    call _atexit
    pop ecx
    ret 0
??_Es@YAXXZ ENDP

```

Листинг 3.107: MSVC 2012: здесь глобальная переменная используется в main()

```

$SG39512 DB 'a string', 00H
$SG39519 DB '%s', 0aH, 00H

_main PROC
    cmp DWORD PTR ?s@3V?$basic_string@DU?$char_traits@D@std@@V?↵
    ↵ $allocator@D@2@std@@A+20, 16
    mov eax, OFFSET ?s@3V?$basic_string@DU?$char_traits@D@std@@V?↵
    ↵ $allocator@D@2@std@@A ;
    S
    cmovae eax, DWORD PTR ?s@3V?$basic_string@DU?$char_traits@D@std@@V?↵
    ↵ $allocator@D@2@std@@A
    push eax
    push OFFSET $SG39519 ; '%s'
    call _printf
    add esp, 8
    xor eax, eax
    ret 0
_main ENDP

```

Листинг 3.108: MSVC 2012: эта функция-деструктор вызывается перед выходом

```

??_Fs@YAXXZ PROC
    push ecx
    cmp DWORD PTR ?s@3V?$basic_string@DU?$char_traits@D@std@@V?↵
    ↵ $allocator@D@2@std@@A+20, 16
    jb SHORT $LN23@dynamic
    push esi
    mov esi, DWORD PTR ?s@3V?$basic_string@DU?$char_traits@D@std@@V?↵
    ↵ $allocator@D@2@std@@A
    lea ecx, DWORD PTR $T2[esp+8]
    call ??0?$Wrap_alloc@V?$allocator@D@std@@@std@@QAE@XZ
    push OFFSET ?s@3V?$basic_string@DU?$char_traits@D@std@@V?↵
    ↵ $allocator@D@2@std@@A ;
    S

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

lea ecx, DWORD PTR $T2[esp+12]
call ??$destroy@PAD@?$_Wrap_alloc@V?$allocator@D@std@@@std@@@QAE@XPAD@Z
lea ecx, DWORD PTR $T1[esp+8]
call ??0?$_Wrap_alloc@V?$allocator@D@std@@@std@@@QAE@XZ
push esi
call ??3@YAXPAX@Z ; operator delete
add esp, 4
pop esi
$LN23@dynamic:
mov DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?↵
↵ $allocator@D@2@@std@@A+20, 15
mov DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?↵
↵ $allocator@D@2@@std@@A+16, 0
mov BYTE PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?↵
↵ $allocator@D@2@@std@@A, 0
pop ecx
ret 0
??_Fs@YAXXZ ENDP

```

В реальности, из **CRT!**, еще до вызова `main()`, вызывается специальная функция, в которой перечислены все конструкторы подобных переменных. Более того: при помощи `atexit()` регистрируется функция, которая будет вызвана в конце работы программы: в этой функции компилятор собирает вызовы деструкторов всех подобных глобальных переменных.

GCC работает похожим образом:

Листинг 3.109: GCC 4.8.1

```

main:
push ebp
mov ebp, esp
and esp, -16
sub esp, 16
mov eax, DWORD PTR s
mov DWORD PTR [esp], eax
call puts
xor eax, eax
leave
ret
.LC0:
.string "a string"
_GLOBAL__sub_I_s:
sub esp, 44
lea eax, [esp+31]
mov DWORD PTR [esp+8], eax
mov DWORD PTR [esp+4], OFFSET FLAT:.LC0
mov DWORD PTR [esp], OFFSET FLAT:s
call _ZN5sC1EPKcRKSaIcE
mov DWORD PTR [esp+8], OFFSET FLAT:__dso_handle
mov DWORD PTR [esp+4], OFFSET FLAT:s
mov DWORD PTR [esp], OFFSET FLAT:_ZN5sD1Ev
call __cxa_atexit
add esp, 44

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

ret
.LFE645:
.size _GLOBAL__sub_I_s, .-_GLOBAL__sub_I_s
.section .init_array,"aw"
.align 4
.long _GLOBAL__sub_I_s
.globl s
.bss
.align 4
.type s, @object
.size s, 4
s:
.zero 4
.hidden __dso_handle

```

Но он не выделяет отдельной функции в которой будут собраны деструкторы: каждый деструктор передается в `atexit()` по одному.

### **std::list**

Хорошо известный всем двусвязный список: каждый элемент имеет два указателя, на следующий и на предыдущий элементы.

Это означает, что расход памяти увеличивается на 2 **слова** на каждый элемент (8 байт в 32-битной среде или 16 байт в 64-битной).

STL в Си++ просто добавляет указатели «next» и «previous» к той вашей структуре, которую вы желаете объединить в список.

Попробуем разобраться с примером в котором простая структура из двух переменных, мы объединим её в список.

Хотя и стандарт Си++ не указывает, как он должен быть реализован, реализации MSVC и GCC простые и похожи друг на друга, так что этот исходный код для обоих:

```

#include <stdio.h>
#include <list>
#include <iostream>

struct a
{
    int x;
    int y;
};

struct List_node
{
    struct List_node* _Next;
    struct List_node* _Prev;
    int x;
    int y;
};

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

void dump_List_node (struct List_node *n)
{
    printf ("ptr=0x%p _Next=0x%p _Prev=0x%p x=%d y=%d\n",
           n, n->_Next, n->_Prev, n->x, n->y);
};

void dump_List_vals (struct List_node* n)
{
    struct List_node* current=n;

    for (;;)
    {
        dump_List_node (current);
        current=current->_Next;
        if (current==n) // end
            break;
    };
};

void dump_List_val (unsigned int *a)
{
#ifdef _MSC_VER
    // в реализации GCC нет поля "size"
    printf ("_Myhead=0x%p, _Mysize=%d\n", a[0], a[1]);
#endif
    dump_List_vals ((struct List_node*)a[0]);
};

int main()
{
    std::list<struct a> l;

    printf ("* empty list:\n");
    dump_List_val((unsigned int*)(void*)&l);

    struct a t1;
    t1.x=1;
    t1.y=2;
    l.push_front (t1);
    t1.x=3;
    t1.y=4;
    l.push_front (t1);
    t1.x=5;
    t1.y=6;
    l.push_back (t1);

    printf ("* 3-elements list:\n");
    dump_List_val((unsigned int*)(void*)&l);

    std::list<struct a>::iterator tmp;
    printf ("node at .begin:\n");
    tmp=l.begin();
    dump_List_node ((struct List_node *)*(void**)&tmp);
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

printf ("node at .end:\n");
tmp=l.end();
dump_List_node ((struct List_node *)*(void*)&tmp);

printf ("* let's count from the beginning:\n");
std::list<struct a>::iterator it=l.begin();
printf ("1st element: %d %d\n", (*it).x, (*it).y);
it++;
printf ("2nd element: %d %d\n", (*it).x, (*it).y);
it++;
printf ("3rd element: %d %d\n", (*it).x, (*it).y);
it++;
printf ("element at .end(): %d %d\n", (*it).x, (*it).y);

printf ("* let's count from the end:\n");
std::list<struct a>::iterator it2=l.end();
printf ("element at .end(): %d %d\n", (*it2).x, (*it2).y);
it2--;
printf ("3rd element: %d %d\n", (*it2).x, (*it2).y);
it2--;
printf ("2nd element: %d %d\n", (*it2).x, (*it2).y);
it2--;
printf ("1st element: %d %d\n", (*it2).x, (*it2).y);

printf ("removing last element...\n");
l.pop_back();
dump_List_val((unsigned int*)(void*)&l);
};

```

## GCC

Начнем с GCC.

При запуске увидим длинный вывод, будем разбирать его по частям.

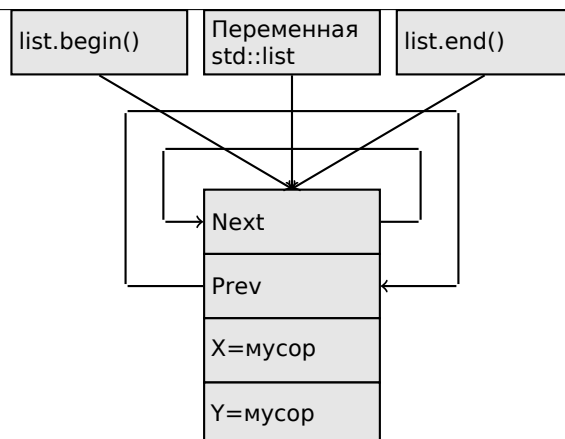
```

* empty list:
ptr=0x0028fe90 _Next=0x0028fe90 _Prev=0x0028fe90 x=3 y=0

```

Видим пустой список. Не смотря на то что он пуст, имеется один элемент с мусором (**АКА!** узел-пустышка (*dummy node*)) в переменных *x* и *y*.

Оба указателя «next» и «prev» указывают на себя:



Это тот момент, когда итераторы `.begin` и `.end` равны друг другу.

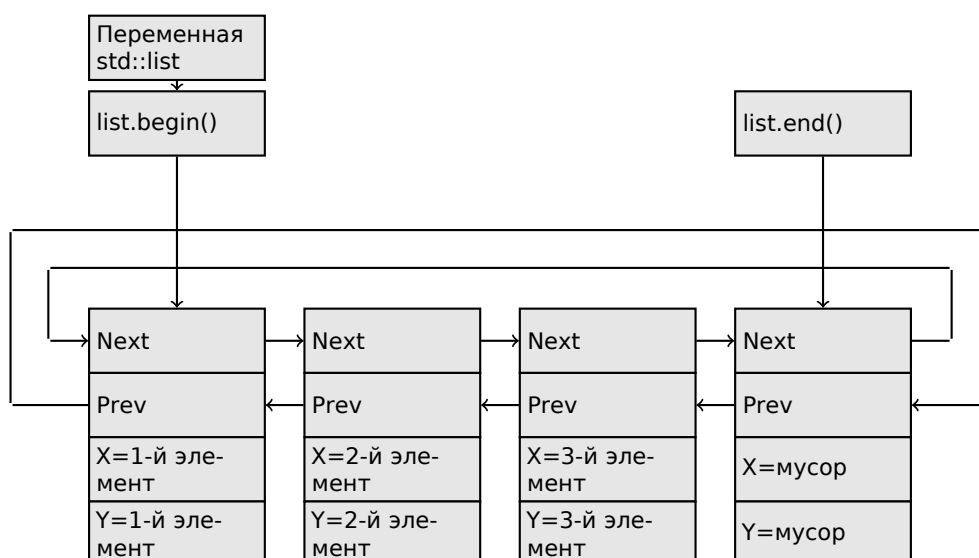
Вставим 3 элемента и список в памяти будет представлен так:

```
* 3-elements list:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x00034b40 _Prev=0x000349a0 x=1 y=2
ptr=0x00034b40 _Next=0x0028fe90 _Prev=0x00034988 x=5 y=6
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
```

Последний элемент всё еще на `0x0028fe90`, он не будет передвинут куда-либо до самого уничтожения списка.

Он все еще содержит случайный мусор в полях  $x$  и  $y$  (5 и 6). Случайно совпало так, что эти значения точно такие же, как и в последнем элементе, но это не значит, что они имеют какое-то значение.

Вот как эти 3 элемента хранятся в памяти:



Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Переменная  $l$  всегда указывает на первый элемент.

Итераторы `.begin()` и `.end()` это не переменные, а функции, возвращающие указатели на соответствующие узлы.

Иметь элемент-пустышку (*dummy node* или *sentinel node*) это очень популярная практика в реализации двусвязных списков.

Без него, многие операции были бы сложнее, и, следовательно, медленнее.

Итератор на самом деле это просто указатель на элемент. `list.begin()` и `list.end()` просто возвращают указатели.

```
node at .begin:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
node at .end:
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
```

Тот факт, что последний элемент имеет указатель на первый и первый имеет указатель на последний, напоминает нам циркулярный список.

Это очень помогает: если иметь указатель только на первый элемент, т.е. тот что в переменной  $l$ , очень легко получить указатель на последний элемент, без необходимости обходить все элементы списка. Вставка элемента в конец списка также быстра благодаря этой особенности.

`operator--` и `operator++` просто выставляют текущее значение итератора на `current_node->prev` или `current_node->next`.

Обратные итераторы (`.rbegin`, `.rend`) работают точно так же, только наоборот.

`operator*` на итераторе просто возвращает указатель на место в структуре, где начинается пользовательская структура, т.е. указатель на самый первый элемент структуры ( $x$ ).

Вставка в список и удаление очень просты: просто выделите новый элемент (или освободите) и исправьте все указатели так, чтобы они были верны.

Вот почему итератор может стать недействительным после удаления элемента: он может всё еще указывать на уже освобожденный элемент.

Это также называется *dangling pointer*. И конечно же, информация из освобожденного элемента, на который указывает итератор, не может использоваться более.

В реализации GCC (по крайней мере 4.8.1) не сохраняется текущая длина списка: это выливается в медленный метод `.size()`: он должен пройти по всему списку считая элементы, просто потому что нет другого способа получить эту информацию. Это означает, что эта операция  $O(n)$ , т.е. она работает тем медленнее, чем больше элементов в списке.

#### Листинг 3.110: Оптимизирующий GCC 4.8.1 `-fno-inline-small-functions`

```
main proc near
  push ebp
  mov  ebp, esp
  push esi
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

push ebx
and esp, 0FFFFFFF0h
sub esp, 20h
lea ebx, [esp+10h]
mov dword ptr [esp], offset s ; "* empty list:"
mov [esp+10h], ebx
mov [esp+14h], ebx
call puts
mov [esp], ebx
call _Z13dump_List_valPj ; dump_List_val(uint *)
lea esi, [esp+18h]
mov [esp+4], esi
mov [esp], ebx
mov dword ptr [esp+18h], 1 ; X нового элемента
mov dword ptr [esp+1Ch], 2 ; Y нового элемента
call _ZNSt4listI1aSaIS0_EE10push_frontERKS0_ ;
std::list<a,std::allocator<a>>::push_front(a const&)
mov [esp+4], esi
mov [esp], ebx
mov dword ptr [esp+18h], 3 ; X нового элемента
mov dword ptr [esp+1Ch], 4 ; Y нового элемента
call _ZNSt4listI1aSaIS0_EE10push_frontERKS0_ ;
std::list<a,std::allocator<a>>::push_front(a const&)
mov dword ptr [esp], 10h
mov dword ptr [esp+18h], 5 ; X нового элемента
mov dword ptr [esp+1Ch], 6 ; Y нового элемента
call _Znwj ; operator new(uint)
cmp eax, 0FFFFFFF8h
jz short loc_80002A6
mov ecx, [esp+1Ch]
mov edx, [esp+18h]
mov [eax+0Ch], ecx
mov [eax+8], edx

loc_80002A6: ; CODE XREF: main+86
mov [esp+4], ebx
mov [esp], eax
call _ZNSt8_detail15_List_node_base7_M_hookEPS0_ ;
std::_detail::_List_node_base::M_hook(std::_detail::_List_node_base*)
mov dword ptr [esp], offset a3ElementsList ; "* 3-elements list:"
call puts
mov [esp], ebx
call _Z13dump_List_valPj ; dump_List_val(uint *)
mov dword ptr [esp], offset aNodeAt_begin ; "node at .begin:"
call puts
mov eax, [esp+10h]
mov [esp], eax
call _Z14dump_List_nodeP9List_node ; dump_List_node(List_node *)
mov dword ptr [esp], offset aNodeAt_end ; "node at .end:"
call puts
mov [esp], ebx
call _Z14dump_List_nodeP9List_node ; dump_List_node(List_node *)
mov dword ptr [esp], offset aLetSCountFromT ; "* let's count from the
beginning:"

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

call puts
mov esi, [esp+10h]
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a1stElementDD ; "1st element: %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov esi, [esi] ; operator++: get ->next pointer
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a2ndElementDD ; "2nd element: %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov esi, [esi] ; operator++: get ->next pointer
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a3rdElementDD ; "3rd element: %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov eax, [esi] ; operator++: get ->next pointer
mov edx, [eax+0Ch]
mov [esp+0Ch], edx
mov eax, [eax+8]
mov dword ptr [esp+4], offset aElementAt_endD ;
"element at .end(): %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov dword ptr [esp], offset aLetSCountFro_0 ; "* let's count from the
end:"
call puts
mov eax, [esp+1Ch]
mov dword ptr [esp+4], offset aElementAt_endD ;
"element at .end(): %d %d\n"
mov dword ptr [esp], 1
mov [esp+0Ch], eax
mov eax, [esp+18h]
mov [esp+8], eax
call __printf_chk
mov esi, [esp+14h]
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a3rdElementDD ; "3rd element: %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov esi, [esi+4] ; operator--: get ->prev pointer

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov  eax, [esi+0Ch]
mov  [esp+0Ch], eax
mov  eax, [esi+8]
mov  dword ptr [esp+4], offset a2ndElementDD ; "2nd element: %d %d\n"
mov  dword ptr [esp], 1
mov  [esp+8], eax
call  __printf_chk
mov  eax, [esi+4] ; operator--: get ->prev pointer
mov  edx, [eax+0Ch]
mov  [esp+0Ch], edx
mov  eax, [eax+8]
mov  dword ptr [esp+4], offset a1stElementDD ; "1st element: %d %d\n"
mov  dword ptr [esp], 1
mov  [esp+8], eax
call  __printf_chk
mov  dword ptr [esp], offset aRemovingLastEl ; "removing last element..."
call  puts
mov  esi, [esp+14h]
mov  [esp], esi
call  _ZNSt8_detail15_List_node_base9_M_unhookEv ;
std::detail::List_node_base::M_unhook(void)
mov  [esp], esi ; void *
call  _ZdlPv ; operator delete(void *)
mov  [esp], ebx
call  _Z13dump_List_valPj ; dump_List_val(uint *)
mov  [esp], ebx
call  _ZNSt10_List_baseI1aSaIS0_EE8_M_clearEv ;
std::List_base<a,std::allocator<a>>::M_clear(void)
lea  esp, [ebp-8]
xor  eax, eax
pop  ebx
pop  esi
pop  ebp
retn
main endp

```

Листинг 3.111: Весь вывод

```

* empty list:
ptr=0x0028fe90 _Next=0x0028fe90 _Prev=0x0028fe90 x=3 y=0
* 3-elements list:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x00034b40 _Prev=0x000349a0 x=1 y=2
ptr=0x00034b40 _Next=0x0028fe90 _Prev=0x00034988 x=5 y=6
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
node at .begin:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
node at .end:
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
* let's count from the beginning:
1st element: 3 4
2nd element: 1 2
3rd element: 5 6
element at .end(): 5 6

```

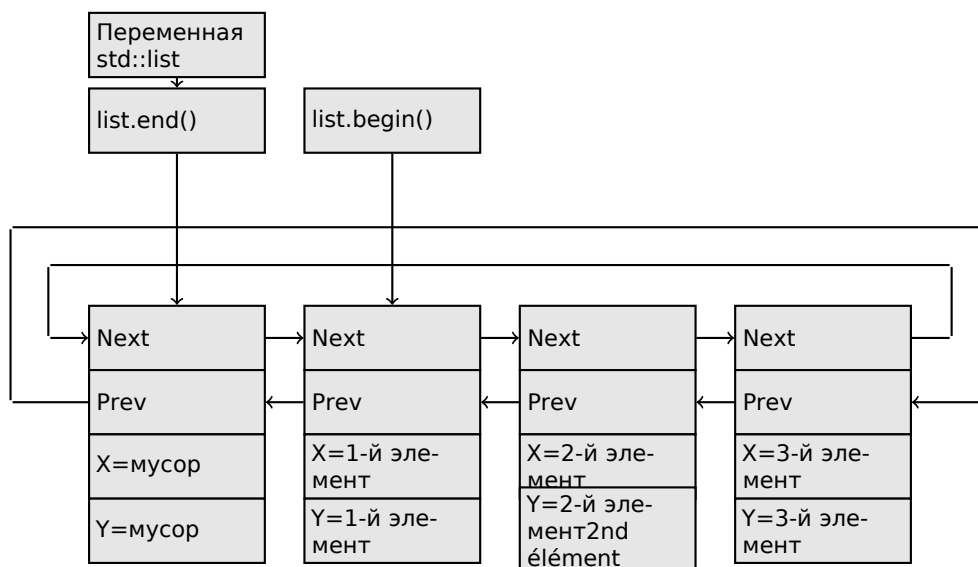
Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
* let's count from the end:
element at .end(): 5 6
3rd element: 5 6
2nd element: 1 2
1st element: 3 4
removing last element...
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x0028fe90 _Prev=0x000349a0 x=1 y=2
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034988 x=5 y=6
```

## MSVC

Реализация MSVC (2012) точно такая же, только еще и сохраняет текущий размер списка. Это означает, что метод `.size()` очень быстр ( $O(1)$ ): просто прочитать одно значение из памяти. С другой стороны, переменная хранящая размер должна корректироваться при каждой вставке/удалении.

Реализация MSVC также немного отлична в смысле расстановки элементов:



У GCC его элемент-пустышка в самом конце списка, а у MSVC в самом начале.

Листинг 3.112: Оптимизирующий MSVC 2012 /Fa2.asm /GS- /Ob1

```
_l$ = -16 ; size = 8
_t1$ = -8 ; size = 8
_main PROC
  sub esp, 16
  push ebx
  push esi
  push edi
  push 0
  push 0
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

lea ecx, DWORD PTR _l$[esp+36]
mov DWORD PTR _l$[esp+40], 0
; выделить первый мусорный элемент
call ?_Buynode0@?$_List_alloc@$0A@U?$_List_base_types@Ua@@V?↵
↳ $allocator@Ua@@@std@@std@@std@@QAEPAU?↵
↳ $_List_node@Ua@@PAX@2@PAU32@0@Z ;
std::_List_alloc<0,std::_List_base_types<a,std::allocator<a> >>::_Buynode0
mov edi, DWORD PTR __imp_printf
mov ebx, eax
push OFFSET $SG40685 ; '* empty list:'
mov DWORD PTR _l$[esp+32], ebx
call edi ; printf
lea eax, DWORD PTR _l$[esp+32]
push eax
call ?dump_List_val@@YAXPAI@Z ; dump_List_val
mov esi, DWORD PTR [ebx]
add esp, 8
lea eax, DWORD PTR _t1$[esp+28]
push eax
push DWORD PTR [esi+4]
lea ecx, DWORD PTR _l$[esp+36]
push esi
mov DWORD PTR _t1$[esp+40], 1 ; данные для нового узла
mov DWORD PTR _t1$[esp+44], 2 ; данные для нового узла
; allocate new node
call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?↵
↳ $allocator@Ua@@@std@@std@@QAEPAU?↵
↳ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ;
std::_List_buy<a,std::allocator<a> >::_Buynode<a const &
mov DWORD PTR [esi+4], eax
mov ecx, DWORD PTR [eax+4]
mov DWORD PTR _t1$[esp+28], 3 ; данные для нового узла
mov DWORD PTR [ecx], eax
mov esi, DWORD PTR [ebx]
lea eax, DWORD PTR _t1$[esp+28]
push eax
push DWORD PTR [esi+4]
lea ecx, DWORD PTR _l$[esp+36]
push esi
mov DWORD PTR _t1$[esp+44], 4 ; данные для нового узла
; allocate new node
call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?↵
↳ $allocator@Ua@@@std@@std@@QAEPAU?↵
↳ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ;
std::_List_buy<a,std::allocator<a> >::_Buynode<a const &
mov DWORD PTR [esi+4], eax
mov ecx, DWORD PTR [eax+4]
mov DWORD PTR _t1$[esp+28], 5 ; данные для нового узла
mov DWORD PTR [ecx], eax
lea eax, DWORD PTR _t1$[esp+28]
push eax
push DWORD PTR [ebx+4]
lea ecx, DWORD PTR _l$[esp+36]
push ebx

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

mov  DWORD PTR _t1$[esp+44], 6 ; данные для нового узла
; allocate new node
call  ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?↵
↳ $allocator@Ua@@@std@@@std@@QAEPAU?↵
↳ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ;
std::_List_buy<a,std::allocator<a> >::_Buynode<a const &
mov  DWORD PTR [ebx+4], eax
mov  ecx, DWORD PTR [eax+4]
push OFFSET $SG40689 ; '* 3-elements list:'
mov  DWORD PTR _l$[esp+36], 3
mov  DWORD PTR [ecx], eax
call  edi ; printf
lea  eax, DWORD PTR _l$[esp+32]
push  eax
call  ?dump_List_val@@YAXPAI@Z ; dump_List_val
push OFFSET $SG40831 ; 'node at .begin:'
call  edi ; printf
push  DWORD PTR [ebx] ; взять поле следующего узла, на который указывает
"\n"
call  ?dump_List_node@@YAXPAUList_node@@@Z ; dump_List_node
push OFFSET $SG40835 ; 'node at .end:'
call  edi ; printf
push  ebx ; pointer to the node / variable points to!
call  ?dump_List_node@@YAXPAUList_node@@@Z ; dump_List_node
push OFFSET $SG40839 ; '* let's count from the begin:'
call  edi ; printf
mov  esi, DWORD PTR [ebx] ; operator++: get ->next pointer
push  DWORD PTR [esi+12]
push  DWORD PTR [esi+8]
push  OFFSET $SG40846 ; '1st element: %d %d'
call  edi ; printf
mov  esi, DWORD PTR [esi] ; operator++: get ->next pointer
push  DWORD PTR [esi+12]
push  DWORD PTR [esi+8]
push  OFFSET $SG40848 ; '2nd element: %d %d'
call  edi ; printf
mov  esi, DWORD PTR [esi] ; operator++: get ->next pointer
push  DWORD PTR [esi+12]
push  DWORD PTR [esi+8]
push  OFFSET $SG40850 ; '3rd element: %d %d'
call  edi ; printf
mov  eax, DWORD PTR [esi] ; operator++: get ->next pointer
add  esp, 64
push  DWORD PTR [eax+12]
push  DWORD PTR [eax+8]
push  OFFSET $SG40852 ; 'element at .end(): %d %d'
call  edi ; printf
push  OFFSET $SG40853 ; '* let's count from the end:'
call  edi ; printf
push  DWORD PTR [ebx+12] ; использовать поля x и y того узла, на который
указывает переменная l
push  DWORD PTR [ebx+8]
push  OFFSET $SG40860 ; 'element at .end(): %d %d'
call  edi ; printf

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov esi, DWORD PTR [ebx+4] ; operator--: get ->prev pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40862 ; '3rd element: %d %d'
call edi ; printf
mov esi, DWORD PTR [esi+4] ; operator--: get ->prev pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40864 ; '2nd element: %d %d'
call edi ; printf
mov eax, DWORD PTR [esi+4] ; operator--: get ->prev pointer
push DWORD PTR [eax+12]
push DWORD PTR [eax+8]
push OFFSET $SG40866 ; '1st element: %d %d'
call edi ; printf
add esp, 64
push OFFSET $SG40867 ; 'removing last element...'
call edi ; printf
mov edx, DWORD PTR [ebx+4]
add esp, 4

; prev=next?
; это единственный элемент, мусор?
; если да, не удаляем его!
cmp edx, ebx
je SHORT $LN349@main
mov ecx, DWORD PTR [edx+4]
mov eax, DWORD PTR [edx]
mov DWORD PTR [ecx], eax
mov ecx, DWORD PTR [edx]
mov eax, DWORD PTR [edx+4]
push edx
mov DWORD PTR [ecx+4], eax
call ???@YAXPAX@Z ; operator delete
add esp, 4
mov DWORD PTR _l$[esp+32], 2
$LN349@main:
lea eax, DWORD PTR _l$[esp+28]
push eax
call ?dump_List_val@@YAXPAI@Z ; dump_List_val
mov eax, DWORD PTR [ebx]
add esp, 4
mov DWORD PTR [ebx], ebx
mov DWORD PTR [ebx+4], ebx
cmp eax, ebx
je SHORT $LN412@main
$LL414@main:
mov esi, DWORD PTR [eax]
push eax
call ???@YAXPAX@Z ; operator delete
add esp, 4
mov eax, esi
cmp esi, ebx

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    jne SHORT $LL414@main
$LN412@main:
    push ebx
    call ??3@YAXPAX@Z ; operator delete
    add esp, 4
    xor eax, eax
    pop edi
    pop esi
    pop ebx
    add esp, 16
    ret 0
_main ENDP

```

В отличие от GCC, код MSVC выделяет элемент-пустышку в самом начале функции при помощи функции «Buynode», она также используется и во время выделения остальных элементов (код GCC выделяет самый первый элемент в локальном стеке).

Листинг 3.113: Весь вывод

```

* empty list:
_Myhead=0x003CC258, _Mysize=0
ptr=0x003CC258 _Next=0x003CC258 _Prev=0x003CC258 x=6226002 y=4522072
* 3-elements list:
_Myhead=0x003CC258, _Mysize=3
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC2A0 x=6226002 y=4522072
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
ptr=0x003CC270 _Next=0x003CC2A0 _Prev=0x003CC288 x=1 y=2
ptr=0x003CC2A0 _Next=0x003CC258 _Prev=0x003CC270 x=5 y=6
node at .begin:
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
node at .end:
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC2A0 x=6226002 y=4522072
* let's count from the beginning:
1st element: 3 4
2nd element: 1 2
3rd element: 5 6
element at .end(): 6226002 4522072
* let's count from the end:
element at .end(): 6226002 4522072
3rd element: 5 6
2nd element: 1 2
1st element: 3 4
removing last element...
_Myhead=0x003CC258, _Mysize=2
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC270 x=6226002 y=4522072
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
ptr=0x003CC270 _Next=0x003CC258 _Prev=0x003CC288 x=1 y=2

```

## C++11 std::forward\_list

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Это то же самое что и `std::list`, но только односвязный список, т.е. имеющий только поле «next» в каждом элементе. Таким образом расход памяти меньше, но возможности идти по списку назад здесь нет.

### **std::vector**

Мы бы назвали `std::vector` «безопасной оболочкой» (wrapper) **PODT!**<sup>34</sup> массива в Си.

Изнутри он очень похож на `std::string` (?? (стр. ??)): он имеет указатель на выделенный буфер, указатель на конец массива и указатель на конец выделенного буфера.

Элементы массива просто лежат в памяти впритык друг к другу, так же, как и в обычном массиве (?? (стр. ??)). В C++11 появился метод `.data()` возвращающий указатель на этот буфер, это похоже на `.c_str()` в `std::string`.

Выделенный буфер в куче может быть больше чем сам массив.

Реализации MSVC и GCC почти одинаковые, отличаются только имена полей в структуре <sup>35</sup>, так что здесь один исходник работающий для обоих компиляторов. И снова здесь Си-подобный код для вывода структуры `std::vector`:

```
#include <stdio.h>
#include <vector>
#include <algorithm>
#include <functional>

struct vector_of_ints
{
    // MSVC names:
    int *Myfirst;
    int *Mylast;
    int *Myend;

    // структура в GCC такая же, а имена там:
    _M_start, _M_finish, _M_end_of_storage
};

void dump(struct vector_of_ints *in)
{
    printf ("_Myfirst=%p, _Mylast=%p, _Myend=%p\n", in->Myfirst, in->Mylast,
    ↵ , in->Myend);
    size_t size=(in->Mylast-in->Myfirst);
    size_t capacity=(in->Myend-in->Myfirst);
    printf ("size=%d, capacity=%d\n", size, capacity);
    for (size_t i=0; i<size; i++)
        printf ("element %d: %d\n", i, in->Myfirst[i]);
};

int main()
```

<sup>34</sup> **PODT!**

<sup>35</sup> внутренности GCC: <http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01371.html>

```

{
    std::vector<int> c;
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(1);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(2);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(3);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(4);
    dump ((struct vector_of_ints*)(void*)&c);
    c.reserve (6);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(5);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(6);
    dump ((struct vector_of_ints*)(void*)&c);
    printf ("%d\n", c.at(5)); // с проверкой границ
    printf ("%d\n", c[8]); // operator[], без проверки границ
};

```

Примерный вывод программы скомпилированной в MSVC:

```

_Myfirst=00000000, _Mylast=00000000, _Myend=00000000
size=0, capacity=0
_Myfirst=0051CF48, _Mylast=0051CF4C, _Myend=0051CF4C
size=1, capacity=1
element 0: 1
_Myfirst=0051CF58, _Mylast=0051CF60, _Myend=0051CF60
size=2, capacity=2
element 0: 1
element 1: 2
_Myfirst=0051C278, _Mylast=0051C284, _Myend=0051C284
size=3, capacity=3
element 0: 1
element 1: 2
element 2: 3
_Myfirst=0051C290, _Mylast=0051C2A0, _Myend=0051C2A0
size=4, capacity=4
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0051B180, _Mylast=0051B190, _Myend=0051B198
size=4, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0051B180, _Mylast=0051B194, _Myend=0051B198
size=5, capacity=6
element 0: 1
element 1: 2
element 2: 3

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

element 3: 4
element 4: 5
_Myfirst=0051B180, _Mylast=0051B198, _Myend=0051B198
size=6, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
element 5: 6
6
6619158

```

Как можно заметить, выделенного буфера в самом начале функции `main()` пока нет. После первого вызова `push_back()` буфер выделяется. И далее, после каждого вызова `push_back()` и длина массива и вместимость буфера (*capacity*) увеличиваются. Но адрес буфера также меняется, потому что вызов функции `push_back()` перевыделяет буфер в *куче* каждый раз. Это дорогая операция, вот почему очень важно предсказать размер будущего массива и зарезервировать место для него при помощи метода `.reserve()`. Самое последнее число — это мусор: там нет элементов массива в этом месте, вот откуда это случайное число. Это иллюстрация того факта что метод `operator[]` в `std::vector` не проверяет индекс на правильность. Более медленный метод `.at()` с другой стороны, проверяет, и подкидывает исключение `std::out_of_range` в случае ошибки.

Давайте посмотрим код:

Листинг 3.114: MSVC 2012 /GS- /Ob1

```

$SG52650 DB '%d', 0aH, 00H
$SG52651 DB '%d', 0aH, 00H

_this$ = -4 ; size = 4
__Pos$ = 8 ; size = 4
?at@$vector@HV?$allocator@H@std@@@std@@@QAEAAHI@Z PROC ;
    std::vector<int,std::allocator<int> >::at, COMDAT
; _this$ = ecx
    push ebp
    mov ebp, esp
    push ecx
    mov DWORD PTR _this$[ebp], ecx
    mov eax, DWORD PTR _this$[ebp]
    mov ecx, DWORD PTR _this$[ebp]
    mov edx, DWORD PTR [eax+4]
    sub edx, DWORD PTR [ecx]
    sar edx, 2
    cmp edx, DWORD PTR __Pos$[ebp]
    ja SHORT $LN1@at
    push OFFSET ??_C@_0BM@NMJKDPP0@invalid?5vector?$DMT?$D0?5subscript?$AA@
    call DWORD PTR __imp_?_Xout_of_range@std@@@YAXPBD@Z
$LN1@at:
    mov eax, DWORD PTR _this$[ebp]
    mov ecx, DWORD PTR [eax]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov     edx, DWORD PTR __Pos$[ebp]
lea    eax, DWORD PTR [ecx+edx*4]
$LN3@at:
mov     esp, ebp
pop     ebp
ret     4
?at@?$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z ENDP ;
std::vector<int,std::allocator<int> >::at

_c$ = -36 ; size = 12
$T1 = -24 ; size = 4
$T2 = -20 ; size = 4
$T3 = -16 ; size = 4
$T4 = -12 ; size = 4
$T5 = -8 ; size = 4
$T6 = -4 ; size = 4
_main PROC
push   ebp
mov    ebp, esp
sub    esp, 36
mov    DWORD PTR _c$[ebp], 0 ; Myfirst
mov    DWORD PTR _c$[ebp+4], 0 ; Mylast
mov    DWORD PTR _c$[ebp+8], 0 ; Myend
lea    eax, DWORD PTR _c$[ebp]
push   eax
call   ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add    esp, 4
mov    DWORD PTR $T6[ebp], 1
lea    ecx, DWORD PTR $T6[ebp]
push   ecx
lea    ecx, DWORD PTR _c$[ebp]
call   ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEAX$QAH@Z ;
std::vector<int,std::allocator<int> >::push_back
lea    edx, DWORD PTR _c$[ebp]
push   edx
call   ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add    esp, 4
mov    DWORD PTR $T5[ebp], 2
lea    eax, DWORD PTR $T5[ebp]
push   eax
lea    ecx, DWORD PTR _c$[ebp]
call   ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEAX$QAH@Z ;
std::vector<int,std::allocator<int> >::push_back
lea    ecx, DWORD PTR _c$[ebp]
push   ecx
call   ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add    esp, 4
mov    DWORD PTR $T4[ebp], 3
lea    edx, DWORD PTR $T4[ebp]
push   edx
lea    ecx, DWORD PTR _c$[ebp]
call   ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEAX$QAH@Z ;
std::vector<int,std::allocator<int> >::push_back
lea    eax, DWORD PTR _c$[ebp]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

push eax
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add esp, 4
mov  DWORD PTR $T3[ebp], 4
lea  ecx, DWORD PTR $T3[ebp]
push ecx
lea  ecx, DWORD PTR _c$[ebp]
call ?push_back@?$vector@HV?$allocator@H@std@@std@@QAEX$$QAH@Z ;
std::vector<int,std::allocator<int> >::push_back
lea  edx, DWORD PTR _c$[ebp]
push edx
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add esp, 4
push 6
lea  ecx, DWORD PTR _c$[ebp]
call ?reserve@?$vector@HV?$allocator@H@std@@std@@QAEXI@Z ;
std::vector<int,std::allocator<int> >::reserve
lea  eax, DWORD PTR _c$[ebp]
push eax
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add esp, 4
mov  DWORD PTR $T2[ebp], 5
lea  ecx, DWORD PTR $T2[ebp]
push ecx
lea  ecx, DWORD PTR _c$[ebp]
call ?push_back@?$vector@HV?$allocator@H@std@@std@@QAEX$$QAH@Z ;
std::vector<int,std::allocator<int> >::push_back
lea  edx, DWORD PTR _c$[ebp]
push edx
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add esp, 4
mov  DWORD PTR $T1[ebp], 6
lea  eax, DWORD PTR $T1[ebp]
push eax
lea  ecx, DWORD PTR _c$[ebp]
call ?push_back@?$vector@HV?$allocator@H@std@@std@@QAEX$$QAH@Z ;
std::vector<int,std::allocator<int> >::push_back
lea  ecx, DWORD PTR _c$[ebp]
push ecx
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add esp, 4
push 5
lea  ecx, DWORD PTR _c$[ebp]
call ?at@?$vector@HV?$allocator@H@std@@std@@QAEEAHI@Z ;
std::vector<int,std::allocator<int> >::at
mov  edx, DWORD PTR [eax]
push edx
push OFFSET $SG52650 ; '%d'
call DWORD PTR __imp__printf
add  esp, 8
mov  eax, 8
shl  eax, 2
mov  ecx, DWORD PTR _c$[ebp]
mov  edx, DWORD PTR [ecx+eax]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

push edx
push OFFSET $SG52651 ; '%d'
call DWORD PTR __imp__printf
add esp, 8
lea ecx, DWORD PTR _c$[ebp]
call ?_Tidy@?$vector@HV?$allocator@H@std@@@std@@IAEXXZ ;
std::vector<int,std::allocator<int> >::_Tidy
xor eax, eax
mov esp, ebp
pop ebp
ret 0
_main ENDP

```

Мы видим, как метод `.at()` проверяет границы и подкидывает исключение в случае ошибки. Число, которое выводит последний вызов `printf()` берется из памяти, без всяких проверок.

Читатель может спросить, почему бы не использовать переменные «size» и «capacity», как это сделано в `std::string`. Должно быть, это для более быстрой проверки границ.

Код генерируемый GCC почти такой же, в целом, но метод `.at()` вставлен прямо в код:

Листинг 3.115: GCC 4.8.1 -fno-inline-small-functions -O1

```

main proc near
push ebp
mov ebp, esp
push edi
push esi
push ebx
and esp, 0FFFFFFF0h
sub esp, 20h
mov dword ptr [esp+14h], 0
mov dword ptr [esp+18h], 0
mov dword ptr [esp+1Ch], 0
lea eax, [esp+14h]
mov [esp], eax
call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov dword ptr [esp+10h], 1
lea eax, [esp+10h]
mov [esp+4], eax
lea eax, [esp+14h]
mov [esp], eax
call _ZNSt6vectorIiSaIiEE9push_backERKi ;
std::vector<int,std::allocator<int>>::push_back(int const&)
lea eax, [esp+14h]
mov [esp], eax
call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov dword ptr [esp+10h], 2
lea eax, [esp+10h]
mov [esp+4], eax
lea eax, [esp+14h]
mov [esp], eax

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

call _ZNSt6vectorIiSaIiEE9push_backERKi ;
std::vector<int,std::allocator<int>>::push_back(int const&)
lea eax, [esp+14h]
mov [esp], eax
call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov dword ptr [esp+10h], 3
lea eax, [esp+10h]
mov [esp+4], eax
lea eax, [esp+14h]
mov [esp], eax
call _ZNSt6vectorIiSaIiEE9push_backERKi ;
std::vector<int,std::allocator<int>>::push_back(int const&)
lea eax, [esp+14h]
mov [esp], eax
call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov dword ptr [esp+10h], 4
lea eax, [esp+10h]
mov [esp+4], eax
lea eax, [esp+14h]
mov [esp], eax
call _ZNSt6vectorIiSaIiEE9push_backERKi ;
std::vector<int,std::allocator<int>>::push_back(int const&)
lea eax, [esp+14h]
mov [esp], eax
call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov ebx, [esp+14h]
mov eax, [esp+1Ch]
sub eax, ebx
cmp eax, 17h
ja short loc_80001CF
mov edi, [esp+18h]
sub edi, ebx
sar edi, 2
mov dword ptr [esp], 18h
call _Znwj ; operator new(uint)
mov esi, eax
test edi, edi
jz short loc_80001AD
lea eax, ds:0[edi*4]
mov [esp+8], eax ; n
mov [esp+4], ebx ; src
mov [esp], esi ; dest
call memmove

loc_80001AD: ; CODE XREF: main+F8
mov eax, [esp+14h]
test eax, eax
jz short loc_80001BD
mov [esp], eax ; void *
call _ZdlPv ; operator delete(void *)

loc_80001BD: ; CODE XREF: main+117
mov [esp+14h], esi
lea eax, [esi+edi*4]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov [esp+18h], eax
add esi, 18h
mov [esp+1Ch], esi

loc_80001CF: ; CODE XREF: main+DD
lea eax, [esp+14h]
mov [esp], eax
call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov dword ptr [esp+10h], 5
lea eax, [esp+10h]
mov [esp+4], eax
lea eax, [esp+14h]
mov [esp], eax
call _ZNSt6vectorIiSaIiEE9push_backERKi ;
std::vector<int,std::allocator<int>>::push_back(int const&)
lea eax, [esp+14h]
mov [esp], eax
call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov dword ptr [esp+10h], 6
lea eax, [esp+10h]
mov [esp+4], eax
lea eax, [esp+14h]
mov [esp], eax
call _ZNSt6vectorIiSaIiEE9push_backERKi ;
std::vector<int,std::allocator<int>>::push_back(int const&)
lea eax, [esp+14h]
mov [esp], eax
call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov eax, [esp+14h]
mov edx, [esp+18h]
sub edx, eax
cmp edx, 17h
ja short loc_8000246
mov dword ptr [esp], offset aVector_m_range ; "vector::_M_range_check"
call _ZSt20__throw_out_of_rangePKc ;
std::_throw_out_of_range(char const*)

loc_8000246: ; CODE XREF: main+19C
mov eax, [eax+14h]
mov [esp+8], eax
mov dword ptr [esp+4], offset aD ; "%d\n"
mov dword ptr [esp], 1
call __printf_chk
mov eax, [esp+14h]
mov eax, [eax+20h]
mov [esp+8], eax
mov dword ptr [esp+4], offset aD ; "%d\n"
mov dword ptr [esp], 1
call __printf_chk
mov eax, [esp+14h]
test eax, eax
jz short loc_80002AC
mov [esp], eax ; void *
call _ZdlPv ; operator delete(void *)

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    jmp short loc_80002AC

    mov ebx, eax
    mov edx, [esp+14h]
    test edx, edx
    jz short loc_80002A4
    mov [esp], edx ; void *
    call _ZdlPv ; operator delete(void *)

loc_80002A4: ; CODE XREF: main+1FE
    mov [esp], ebx
    call _Unwind_Resume

loc_80002AC: ; CODE XREF: main+1EA
    ; main+1F4
    mov eax, 0
    lea esp, [ebp-0Ch]
    pop ebx
    pop esi
    pop edi
    pop ebp

locret_80002B8: ; DATA XREF: .eh_frame:08000510
    ; .eh_frame:080005BC
    retn
main endp

```

Метод `.reserve()` точно так же вставлен прямо в код `main()`. Он вызывает `new()` если буфер слишком мал для нового массива, вызывает `memmove()` для копирования содержимого буфера, и вызывает `delete()` для освобождения старого буфера.

Посмотрим, что выводит программа будучи скомпилированная GCC:

```

_Myfirst=0x(nil), _Mylast=0x(nil), _Myend=0x(nil)
size=0, capacity=0
_Myfirst=0x8257008, _Mylast=0x825700c, _Myend=0x825700c
size=1, capacity=1
element 0: 1
_Myfirst=0x8257018, _Mylast=0x8257020, _Myend=0x8257020
size=2, capacity=2
element 0: 1
element 1: 2
_Myfirst=0x8257028, _Mylast=0x8257034, _Myend=0x8257038
size=3, capacity=4
element 0: 1
element 1: 2
element 2: 3
_Myfirst=0x8257028, _Mylast=0x8257038, _Myend=0x8257038
size=4, capacity=4
element 0: 1
element 1: 2
element 2: 3

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

element 3: 4
_Myfirst=0x8257040, _Mylast=0x8257050, _Myend=0x8257058
size=4, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0x8257040, _Mylast=0x8257054, _Myend=0x8257058
size=5, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
_Myfirst=0x8257040, _Mylast=0x8257058, _Myend=0x8257058
size=6, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
element 5: 6
6
0

```

Мы можем заметить, что буфер растёт иначе чем в MSVC.

При помощи простых экспериментов становится ясно, что в реализации MSVC буфер увеличивается на ~50% каждый раз, когда он должен был увеличен, а у GCC он увеличивается на 100% каждый раз, т.е. удваивается.

### **std::map и std::set**

Двоичное дерево—это ещё одна фундаментальная структура данных. Как следует из названия, это дерево, но у каждого узла максимум 2 связи с другими узлами. Каждый узел имеет ключ и/или значение: в `std::set` у каждого узла есть ключ, в `std::map` у каждого узла есть и ключ и значение.

Обычно, именно при помощи двоичных деревьев реализуются «словари» пар ключ-значения (**АКА!** «ассоциативные массивы»).

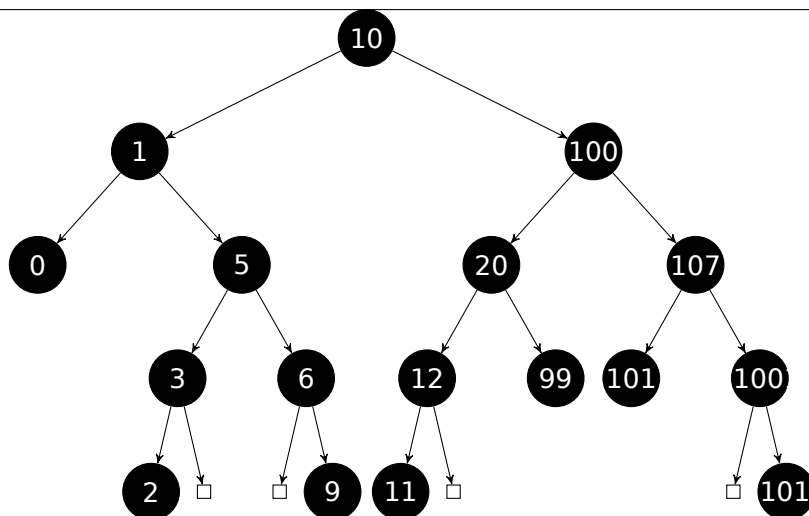
Двоичные деревья имеют по крайней мере три важных свойства:

- Все ключи всегда хранятся в отсортированном виде.
- Могут храниться ключи любых типов. Алгоритмы для работы с двоичными деревьями не зависят от типа ключа, для работы им нужна только функция для сравнения ключей.
- Поиск заданного ключа относительно быстрый по сравнению со списками или массивами.

Очень простой пример: давайте сохраним вот эти числа в двоичном дереве: 0, 1, 2, 3, 5, 6, 9, 10, 11, 12, 20, 99, 100, 101, 107, 1001, 1010.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



Все ключи меньше чем значение ключа узла, сохраняются по левой стороне. Все ключи больше чем значение ключа узла, сохраняются по правой стороне.

Таким образом, алгоритм для поиска нужного ключа прост: если искомое значение меньше чем значение текущего узла: двигаемся влево, если больше: двигаемся вправо, останавливаемся если они равны. Таким образом, алгоритм может искать числа, текстовые строки, и т. д., пользуясь только функцией сравнения ключей.

Все ключи имеют уникальные значения.

Учитывая это, нужно  $\approx \log_2 n$  шагов для поиска ключа в сбалансированном дереве, содержащем  $n$  ключей. Это  $\approx 10$  шагов для  $\approx 1000$  ключей, или  $\approx 13$  шагов для  $\approx 10000$  ключей. Неплохо, но для этого дерево всегда должно быть сбалансировано: т.е. ключи должны быть равномерно распределены на всех ярусах. Операции вставки и удаления проводят дополнительную работу по обслуживанию дерева и сохранения его в сбалансированном состоянии.

Известно несколько популярных алгоритмов балансировки, включая AVL-деревья и красно-черные деревья. Последний дополняет узел значением «цвета» для упрощения балансировки, таким образом каждый узел может быть «красным» или «черным».

Реализации `std::map` и `std::set` обоих GCC и MSVC используют красно-черные деревья.

`std::set` содержит только ключи. `std::map` это «расширенная» версия `set`: здесь имеется еще и значение (`value`) на каждом узле.

## MSVC

```

#include <map>
#include <set>
#include <string>
#include <iostream>
  
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

else
    printf ("%d [%s]\n", n->first, n->second);
if (n->Left->Isnll==0)
{
    printf ("%.*sL-----", tabs, ALOT_OF_TABS);
    dump_as_tree (tabs+1, n->Left, is_set);
};
if (n->Right->Isnll==0)
{
    printf ("%.*sR-----", tabs, ALOT_OF_TABS);
    dump_as_tree (tabs+1, n->Right, is_set);
};
};

void dump_map_and_set(struct tree_struct *m, bool is_set)
{
    printf ("ptr=0x%p, Myhead=0x%p, Mysize=%d\n", m, m->Myhead, m->Mysize);
    dump_tree_node (m->Myhead, is_set, true);
    printf ("As a tree:\n");
    printf ("root----");
    dump_as_tree (1, m->Myhead->Parent, is_set);
};

int main()
{
    // map

    std::map<int, const char*> m;

    m[10]="ten";
    m[20]="twenty";
    m[3]="three";
    m[101]="one hundred one";
    m[100]="one hundred";
    m[12]="twelve";
    m[107]="one hundred seven";
    m[0]="zero";
    m[1]="one";
    m[6]="six";
    m[99]="ninety-nine";
    m[5]="five";
    m[11]="eleven";
    m[1001]="one thousand one";
    m[1010]="one thousand ten";
    m[2]="two";
    m[9]="nine";
    printf ("dumping m as map:\n");
    dump_map_and_set ((struct tree_struct *) (void*)&m, false);

    std::map<int, const char*>::iterator it1=m.begin();
    printf ("m.begin():\n");
    dump_tree_node ((struct tree_node *) (void*)&it1, false, false);
    it1=m.end();
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

printf ("m.end():\n");
dump_tree_node ((struct tree_node *)*(void*)&it1, false, false);

// set

std::set<int> s;
s.insert(123);
s.insert(456);
s.insert(11);
s.insert(12);
s.insert(100);
s.insert(1001);
printf ("dumping s as set:\n");
dump_map_and_set ((struct tree_struct *)*(void*)&s, true);
std::set<int>::iterator it2=s.begin();
printf ("s.begin():\n");
dump_tree_node ((struct tree_node *)*(void*)&it2, true, false);
it2=s.end();
printf ("s.end():\n");
dump_tree_node ((struct tree_node *)*(void*)&it2, true, false);
};

```

## Листинг 3.116: MSVC 2012

```

dumping m as map:
ptr=0x0020FE04, Myhead=0x005BB3A0, Mysize=17
ptr=0x005BB3A0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0x005BB580 Color=1 ↗
  ↳ Isn1l=1
ptr=0x005BB3C0 Left=0x005BB4C0 Parent=0x005BB3A0 Right=0x005BB440 Color=1 ↗
  ↳ Isn1l=0
first=10 second=[ten]
ptr=0x005BB4C0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0x005BB520 Color=1 ↗
  ↳ Isn1l=0
first=1 second=[one]
ptr=0x005BB4A0 Left=0x005BB3A0 Parent=0x005BB4C0 Right=0x005BB3A0 Color=1 ↗
  ↳ Isn1l=0
first=0 second=[zero]
ptr=0x005BB520 Left=0x005BB400 Parent=0x005BB4C0 Right=0x005BB4E0 Color=0 ↗
  ↳ Isn1l=0
first=5 second=[five]
ptr=0x005BB400 Left=0x005BB5A0 Parent=0x005BB520 Right=0x005BB3A0 Color=1 ↗
  ↳ Isn1l=0
first=3 second=[three]
ptr=0x005BB5A0 Left=0x005BB3A0 Parent=0x005BB400 Right=0x005BB3A0 Color=0 ↗
  ↳ Isn1l=0
first=2 second=[two]
ptr=0x005BB4E0 Left=0x005BB3A0 Parent=0x005BB520 Right=0x005BB5C0 Color=1 ↗
  ↳ Isn1l=0
first=6 second=[six]
ptr=0x005BB5C0 Left=0x005BB3A0 Parent=0x005BB4E0 Right=0x005BB3A0 Color=0 ↗
  ↳ Isn1l=0
first=9 second=[nine]
ptr=0x005BB440 Left=0x005BB3E0 Parent=0x005BB3C0 Right=0x005BB480 Color=1 ↗
  ↳ Isn1l=0

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

first=100 second=[one hundred]
ptr=0x005BB3E0 Left=0x005BB460 Parent=0x005BB440 Right=0x005BB500 Color=0 ↗
  ↳ Isn1l=0
first=20 second=[twenty]
ptr=0x005BB460 Left=0x005BB540 Parent=0x005BB3E0 Right=0x005BB3A0 Color=1 ↗
  ↳ Isn1l=0
first=12 second=[twelve]
ptr=0x005BB540 Left=0x005BB3A0 Parent=0x005BB460 Right=0x005BB3A0 Color=0 ↗
  ↳ Isn1l=0
first=11 second=[eleven]
ptr=0x005BB500 Left=0x005BB3A0 Parent=0x005BB3E0 Right=0x005BB3A0 Color=1 ↗
  ↳ Isn1l=0
first=99 second=[ninety-nine]
ptr=0x005BB480 Left=0x005BB420 Parent=0x005BB440 Right=0x005BB560 Color=0 ↗
  ↳ Isn1l=0
first=107 second=[one hundred seven]
ptr=0x005BB420 Left=0x005BB3A0 Parent=0x005BB480 Right=0x005BB3A0 Color=1 ↗
  ↳ Isn1l=0
first=101 second=[one hundred one]
ptr=0x005BB560 Left=0x005BB3A0 Parent=0x005BB480 Right=0x005BB580 Color=1 ↗
  ↳ Isn1l=0
first=1001 second=[one thousand one]
ptr=0x005BB580 Left=0x005BB3A0 Parent=0x005BB560 Right=0x005BB3A0 Color=0 ↗
  ↳ Isn1l=0
first=1010 second=[one thousand ten]
As a tree:
root----10 [ten]
  L-----1 [one]
    L-----0 [zero]
    R-----5 [five]
      L-----3 [three]
        L-----2 [two]
        R-----6 [six]
          R-----9 [nine]
  R-----100 [one hundred]
    L-----20 [twenty]
      L-----12 [twelve]
        L-----11 [eleven]
        R-----99 [ninety-nine]
      R-----107 [one hundred seven]
        L-----101 [one hundred one]
        R-----1001 [one thousand one]
          R-----1010 [one thousand ten]

m.begin():
ptr=0x005BB4A0 Left=0x005BB3A0 Parent=0x005BB4C0 Right=0x005BB3A0 Color=1 ↗
  ↳ Isn1l=0
first=0 second=[zero]
m.end():
ptr=0x005BB3A0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0x005BB580 Color=1 ↗
  ↳ Isn1l=1

dumping s as set:
ptr=0x0020FDFC, Myhead=0x005BB5E0, Mysize=6

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

ptr=0x005BB5E0 Left=0x005BB640 Parent=0x005BB600 Right=0x005BB6A0 Color=1 ↗
  ↳ Isn1l=1
ptr=0x005BB600 Left=0x005BB660 Parent=0x005BB5E0 Right=0x005BB620 Color=1 ↗
  ↳ Isn1l=0
first=123
ptr=0x005BB660 Left=0x005BB640 Parent=0x005BB600 Right=0x005BB680 Color=1 ↗
  ↳ Isn1l=0
first=12
ptr=0x005BB640 Left=0x005BB5E0 Parent=0x005BB660 Right=0x005BB5E0 Color=0 ↗
  ↳ Isn1l=0
first=11
ptr=0x005BB680 Left=0x005BB5E0 Parent=0x005BB660 Right=0x005BB5E0 Color=0 ↗
  ↳ Isn1l=0
first=100
ptr=0x005BB620 Left=0x005BB5E0 Parent=0x005BB600 Right=0x005BB6A0 Color=1 ↗
  ↳ Isn1l=0
first=456
ptr=0x005BB6A0 Left=0x005BB5E0 Parent=0x005BB620 Right=0x005BB5E0 Color=0 ↗
  ↳ Isn1l=0
first=1001
As a tree:
root----123
      L-----12
            L-----11
            R-----100
      R-----456
            R-----1001
s.begin():
ptr=0x005BB640 Left=0x005BB5E0 Parent=0x005BB660 Right=0x005BB5E0 Color=0 ↗
  ↳ Isn1l=0
first=11
s.end():
ptr=0x005BB5E0 Left=0x005BB640 Parent=0x005BB600 Right=0x005BB6A0 Color=1 ↗
  ↳ Isn1l=1

```

Структура не запакована, так что оба значения типа *char* занимают по 4 байта.

В `std::map`, `first` и `second` могут быть представлены как одно значение типа `std::pair`. `std::set` имеет только одно значение в этом месте структуры.

Текущий размер дерева всегда присутствует, как и в случае реализации `std::list` в MSVC (?? (стр. ??)).

Как и в случае с `std::list`, итераторы это просто указатели на узлы. Итератор `.begin()` указывает на минимальный ключ. Этот указатель нигде не сохранен (как в списках), минимальный ключ дерева нужно находить каждый раз. `operator--` и `operator++` перемещают указатель не текущий узел на узел-предшественник или узел-преемник, т.е. узлы содержащие предыдущий и следующий ключ. Алгоритмы для всех этих операций описаны в [Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford, *Introduction to Algorithms, Third Edition*, (2009)].

Итератор `.end()` указывает на узел-пустышку, он имеет 1 в `Isn1l`, что означает, что у узла нет ключа и/или значения.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Так что его можно рассматривать как «landing zone» в **HDD!**<sup>36</sup>. Этот узел часто называется *sentinel* [см. N. Wirth, *Algorithms and Data Structures*, 1985]<sup>37</sup>.

Поле «parent» узла-пустышки указывает на корневой узел, который служит как вершина дерева, и уже содержит информацию.

## GCC

```
#include <stdio.h>
#include <map>
#include <set>
#include <string>
#include <iostream>

struct map_pair
{
    int key;
    const char *value;
};

struct tree_node
{
    int M_color; // 0 - Red, 1 - Black
    struct tree_node *M_parent;
    struct tree_node *M_left;
    struct tree_node *M_right;
};

struct tree_struct
{
    int M_key_compare;
    struct tree_node M_header;
    size_t M_node_count;
};

void dump_tree_node (struct tree_node *n, bool is_set, bool traverse, bool ↵
↳ dump_keys_and_values)
{
    printf ("ptr=0x%p M_left=0x%p M_parent=0x%p M_right=0x%p M_color=%d\n",
           n, n->M_left, n->M_parent, n->M_right, n->M_color);

    void *point_after_struct=((char*)n)+sizeof(struct tree_node);

    if (dump_keys_and_values)
    {
        if (is_set)
            printf ("key=%d\n", *(int*)point_after_struct);
        else
        {
            struct map_pair *p=(struct map_pair *)point_after_struct;

```

<sup>36</sup>**HDD!**

<sup>37</sup><http://www.ethoberon.ethz.ch/WirthPubl/AD.pdf>

```

        printf ("key=%d value=[%s]\n", p->key, p->value);
    };
};

if (traverse==false)
    return;

if (n->M_left)
    dump_tree_node (n->M_left, is_set, traverse, dump_keys_and_values);
if (n->M_right)
    dump_tree_node (n->M_right, is_set, traverse, dump_keys_and_values)↵
↵ ;
};

const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n, bool is_set)
{
    void *point_after_struct=((char*)n)+sizeof(struct tree_node);

    if (is_set)
        printf ("%d\n", *(int*)point_after_struct);
    else
    {
        struct map_pair *p=(struct map_pair *)point_after_struct;
        printf ("%d [%s]\n", p->key, p->value);
    }

    if (n->M_left)
    {
        printf ("%.*sL-----", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_left, is_set);
    };
    if (n->M_right)
    {
        printf ("%.*sR-----", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_right, is_set);
    };
};

void dump_map_and_set(struct tree_struct *m, bool is_set)
{
    printf ("ptr=0x%p, M_key_compare=0x%x, M_header=0x%p, M_node_count=%d\n↵
↵ ",
        m, m->M_key_compare, &m->M_header, m->M_node_count);
    dump_tree_node (m->M_header.M_parent, is_set, true, true);
    printf ("As a tree:\n");
    printf ("root----");
    dump_as_tree (1, m->M_header.M_parent, is_set);
};

int main()
{

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

// map

std::map<int, const char*> m;

m[10]="ten";
m[20]="twenty";
m[3]="three";
m[101]="one hundred one";
m[100]="one hundred";
m[12]="twelve";
m[107]="one hundred seven";
m[0]="zero";
m[1]="one";
m[6]="six";
m[99]="ninety-nine";
m[5]="five";
m[11]="eleven";
m[1001]="one thousand one";
m[1010]="one thousand ten";
m[2]="two";
m[9]="nine";

printf ("dumping m as map:\n");
dump_map_and_set ((struct tree_struct *) (void*)&m, false);

std::map<int, const char*>::iterator it1=m.begin();
printf ("m.begin():\n");
dump_tree_node ((struct tree_node *) (void*)&it1, false, false, true);
it1=m.end();
printf ("m.end():\n");
dump_tree_node ((struct tree_node *) (void*)&it1, false, false, false)↵
↵ ;

// set

std::set<int> s;
s.insert(123);
s.insert(456);
s.insert(11);
s.insert(12);
s.insert(100);
s.insert(1001);
printf ("dumping s as set:\n");
dump_map_and_set ((struct tree_struct *) (void*)&s, true);
std::set<int>::iterator it2=s.begin();
printf ("s.begin():\n");
dump_tree_node ((struct tree_node *) (void*)&it2, true, false, true);
it2=s.end();
printf ("s.end():\n");
dump_tree_node ((struct tree_node *) (void*)&it2, true, false, false);
};

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

## Листинг 3.117: GCC 4.8.1

```

dumping m as map:
ptr=0x0028FE3C, M_key_compare=0x402b70, M_header=0x0028FE40, M_node_count ↗
↳ =17
ptr=0x007A4988 M_left=0x007A4C00 M_parent=0x0028FE40 M_right=0x007A4B80 ↗
↳ M_color=1
key=10 value=[ten]
ptr=0x007A4C00 M_left=0x007A4BE0 M_parent=0x007A4988 M_right=0x007A4C60 ↗
↳ M_color=1
key=1 value=[one]
ptr=0x007A4BE0 M_left=0x00000000 M_parent=0x007A4C00 M_right=0x00000000 ↗
↳ M_color=1
key=0 value=[zero]
ptr=0x007A4C60 M_left=0x007A4B40 M_parent=0x007A4C00 M_right=0x007A4C20 ↗
↳ M_color=0
key=5 value=[five]
ptr=0x007A4B40 M_left=0x007A4CE0 M_parent=0x007A4C60 M_right=0x00000000 ↗
↳ M_color=1
key=3 value=[three]
ptr=0x007A4CE0 M_left=0x00000000 M_parent=0x007A4B40 M_right=0x00000000 ↗
↳ M_color=0
key=2 value=[two]
ptr=0x007A4C20 M_left=0x00000000 M_parent=0x007A4C60 M_right=0x007A4D00 ↗
↳ M_color=1
key=6 value=[six]
ptr=0x007A4D00 M_left=0x00000000 M_parent=0x007A4C20 M_right=0x00000000 ↗
↳ M_color=0
key=9 value=[nine]
ptr=0x007A4B80 M_left=0x007A49A8 M_parent=0x007A4988 M_right=0x007A4BC0 ↗
↳ M_color=1
key=100 value=[one hundred]
ptr=0x007A49A8 M_left=0x007A4BA0 M_parent=0x007A4B80 M_right=0x007A4C40 ↗
↳ M_color=0
key=20 value=[twenty]
ptr=0x007A4BA0 M_left=0x007A4C80 M_parent=0x007A49A8 M_right=0x00000000 ↗
↳ M_color=1
key=12 value=[twelve]
ptr=0x007A4C80 M_left=0x00000000 M_parent=0x007A4BA0 M_right=0x00000000 ↗
↳ M_color=0
key=11 value=[eleven]
ptr=0x007A4C40 M_left=0x00000000 M_parent=0x007A49A8 M_right=0x00000000 ↗
↳ M_color=1
key=99 value=[ninety-nine]
ptr=0x007A4BC0 M_left=0x007A4B60 M_parent=0x007A4B80 M_right=0x007A4CA0 ↗
↳ M_color=0
key=107 value=[one hundred seven]
ptr=0x007A4B60 M_left=0x00000000 M_parent=0x007A4BC0 M_right=0x00000000 ↗
↳ M_color=1
key=101 value=[one hundred one]
ptr=0x007A4CA0 M_left=0x00000000 M_parent=0x007A4BC0 M_right=0x007A4CC0 ↗
↳ M_color=1
key=1001 value=[one thousand one]
ptr=0x007A4CC0 M_left=0x00000000 M_parent=0x007A4CA0 M_right=0x00000000 ↗

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    ↪ M_color=0
key=1010 value=[one thousand ten]
As a tree:
root----10 [ten]
    L-----1 [one]
        L-----0 [zero]
        R-----5 [five]
            L-----3 [three]
                L-----2 [two]
                R-----6 [six]
                    R-----9 [nine]
R-----100 [one hundred]
    L-----20 [twenty]
        L-----12 [twelve]
            L-----11 [eleven]
            R-----99 [ninety-nine]
R-----107 [one hundred seven]
    L-----101 [one hundred one]
    R-----1001 [one thousand one]
        R-----1010 [one thousand ten]

m.begin():
ptr=0x007A4BE0 M_left=0x00000000 M_parent=0x007A4C00 M_right=0x00000000 ↵
    ↪ M_color=1
key=0 value=[zero]
m.end():
ptr=0x0028FE40 M_left=0x007A4BE0 M_parent=0x007A4988 M_right=0x007A4CC0 ↵
    ↪ M_color=0

dumping s as set:
ptr=0x0028FE20, M_key_compare=0x8, M_header=0x0028FE24, M_node_count=6
ptr=0x007A1E80 M_left=0x01D5D890 M_parent=0x0028FE24 M_right=0x01D5D850 ↵
    ↪ M_color=1
key=123
ptr=0x01D5D890 M_left=0x01D5D870 M_parent=0x007A1E80 M_right=0x01D5D8B0 ↵
    ↪ M_color=1
key=12
ptr=0x01D5D870 M_left=0x00000000 M_parent=0x01D5D890 M_right=0x00000000 ↵
    ↪ M_color=0
key=11
ptr=0x01D5D8B0 M_left=0x00000000 M_parent=0x01D5D890 M_right=0x00000000 ↵
    ↪ M_color=0
key=100
ptr=0x01D5D850 M_left=0x00000000 M_parent=0x007A1E80 M_right=0x01D5D8D0 ↵
    ↪ M_color=1
key=456
ptr=0x01D5D8D0 M_left=0x00000000 M_parent=0x01D5D850 M_right=0x00000000 ↵
    ↪ M_color=0
key=1001
As a tree:
root----123
    L-----12
        L-----11
        R-----100

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

R-----456
R-----1001
s.begin():
ptr=0x01D5D870 M_left=0x00000000 M_parent=0x01D5D890 M_right=0x00000000 ↗
  ↘ M_color=0
key=11
s.end():
ptr=0x0028FE24 M_left=0x01D5D870 M_parent=0x007A1E80 M_right=0x01D5D8D0 ↗
  ↘ M_color=0

```

Реализация в GCC очень похожа <sup>38</sup>. Разница только в том, что здесь нет поля `Isnil`, так что структура занимает немного меньше места в памяти чем та что реализована в MSVC.

Узел-пустышка — это также место, куда указывает итератор `.end()`, не имеющий ключа и/или значения.

### Демонстрация перебалансировки (GCC)

Вот также демонстрация, показывающая нам как дерево может перебалансироваться после вставок.

Листинг 3.118: GCC

```

#include <stdio.h>
#include <map>
#include <set>
#include <string>
#include <iostream>

struct map_pair
{
    int key;
    const char *value;
};

struct tree_node
{
    int M_color; // 0 - Red, 1 - Black
    struct tree_node *M_parent;
    struct tree_node *M_left;
    struct tree_node *M_right;
};

struct tree_struct
{
    int M_key_compare;
    struct tree_node M_header;
    size_t M_node_count;
};

```

<sup>38</sup>[http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.1/stl\\_\\_tree\\_8h-source.html](http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.1/stl__tree_8h-source.html)

```
const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n)
{
    void *point_after_struct=((char*)n)+sizeof(struct tree_node);

    printf ("%d\n", *(int*)point_after_struct);

    if (n->M_left)
    {
        printf (".*sL-----", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_left);
    };
    if (n->M_right)
    {
        printf (".*sR-----", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_right);
    };
};

void dump_map_and_set(struct tree_struct *m)
{
    printf ("root----");
    dump_as_tree (1, m->M_header.M_parent);
};

int main()
{
    std::set<int> s;
    s.insert(123);
    s.insert(456);
    printf ("123, 456 has been inserted\n");
    dump_map_and_set ((struct tree_struct *) (void*)&s);
    s.insert(11);
    s.insert(12);
    printf ("\n");
    printf ("11, 12 has been inserted\n");
    dump_map_and_set ((struct tree_struct *) (void*)&s);
    s.insert(100);
    s.insert(1001);
    printf ("\n");
    printf ("100, 1001 has been inserted\n");
    dump_map_and_set ((struct tree_struct *) (void*)&s);
    s.insert(667);
    s.insert(1);
    s.insert(4);
    s.insert(7);
    printf ("\n");
    printf ("667, 1, 4, 7 has been inserted\n");
    dump_map_and_set ((struct tree_struct *) (void*)&s);
    printf ("\n");
};
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Листинг 3.119: GCC 4.8.1

```

123, 456 has been inserted
root----123
      R-----456

11, 12 has been inserted
root----123
      L-----11
            R-----12
      R-----456

100, 1001 has been inserted
root----123
      L-----12
            L-----11
            R-----100
      R-----456
            R-----1001

667, 1, 4, 7 has been inserted
root----12
      L-----4
            L-----1
            R-----11
                  L-----7
      R-----123
            L-----100
            R-----667
                  L-----456
                  R-----1001

```

### 3.19.5. Память

Иногда вы можете услышать от программистов на Си++ «выделить память на/в стеке» и/или «выделить память в [куче](#)».

Выделение памяти *на стеке*:

```

void f()
{
    ...

    Class o=Class(...);

    ...
};

```

Память для объекта (или структуры) выделяется в стеке, при помощи простого сдвига **SP!**. Память освобождается во время выхода из ф-ции, или, более точно, в конце *области видимости (scope)*—**SP!** возвращается в своё состояние (такое же, как при старте ф-ции) и вызывается деструктор класса *Class*. В такой же манере, выделяется и освобождается память для структуры в Си.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Выделение памяти для объекта в *куче*:

```
void f1()
{
    ...
    Class *o=new Class(...);
    ...
};

void f2()
{
    ...
    delete o;
    ...
};
```

Это то же самое, как и выделять память для структуры используя ф-цию *malloc()*. На самом деле, *new* в Си++ это *wrapper* для *malloc()*, а *delete* это *wrapper* для *free()*. Т.к., блок памяти был выделен в *куче*, он должен быть освобожден явно, используя *delete*. Деструктор класса будет автоматически вызван прямо перед этим моментом.

Какой метод лучше? Выделение *на стеке* очень быстрое и подходит для маленьких объектов с коротким сроком жизни, которые будут использоваться только в текущей ф-ции.

Выделение *в куче* медленнее, и лучше для объектов с долгим сроком жизни, которые будут использоваться в нескольких (или многих) ф-циях. Также, объекты выделенные в *куче* подвержены утечкам памяти, потому что их нужно освободить явно, но об этом легко забыть.

Так или иначе, это дело вкуса.

## 3.20. Отрицательные индексы массивов

Возможно адресовать место в памяти *перед* массивом задавая отрицательный индекс, например, *array[-1]*.

### 3.20.1. Адресация строки с конца

**PL!** Питон позволяет адресовать строки с конца. Например, *string[-1]* возвращает последний символ, *string[-2]* возвращает предпоследний, и т. д. Трудно поверить, но в Си/Си++ это также возможно:

```
#include <string.h>
#include <stdio.h>

int main()
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

{
    char *s="Hello, world!";
    char *s_end=s+strlen(s);

    printf ("last character: %c\n", s_end[-1]);
    printf ("penultimate character: %c\n", s_end[-2]);
};

```

Это работает, но *s\_end* должен всегда содержать адрес оконечивающего нулевого байта строки *s*. Если длина строки *s* изменилась, *s\_end* должен обновиться. Это сомнительный трюк, но опять же, это хорошая демонстрация отрицательных индексов.

### 3.20.2. Адресация некоторого блока с конца

Вначале вспомним, почему стек растет в обратную сторону (?? (стр. ??)). Есть в памяти какой-то блок и вам нужно держать там и кучу (*heap*) и стек, и вы не уверены, насколько вырастут обе структуры во время исполнения кода.

Вы можете установить указатель *heap* в начало блока, затем установить указатель *stack* в конец блока (*heap + size\_of\_block*), затем вы можете адресовать *n*-ый элемент стека как *stack[-n]*. Например, *stack[-1]* для 1-го элемента, *stack[-2]* для 2-го, и т. д.

Это работает точно так же, как и трюк с адресацией строки с конца.

Проверять, не пересекаются ли структуры друг с другом легко: просто убедиться, что адрес последнего элемента в *heap* всегда меньше, чем адрес последнего элемента в *stack*.

К сожалению, индекс  $-0$  работать не будет, т.к. способ представления отрицательных чисел (*дополнительный код*) не поддерживает отрицательный ноль, так что он не будет отличим от положительного нуля.

Этот метод также упоминается в "Transaction processing" Jim Gray, 1993, глава "The Tuple-Oriented File System", стр. 755.

### 3.20.3. Массивы начинающиеся с 1

В Фортране и Mathematica первый элемент массива адресуется как 1-ый, вероятно, потому что так традиционно в математике. Другие **PL!** как Си/Си++ адресуют его как 0-й. Как лучше? Эдсгер Дейкстра считал что последний способ лучше <sup>39</sup>.

Но привычка у программистов после Фортрана может остаться, так что все еще возможно адресовать первый элемент через 1 в Си/Си++используя этот трюк:

```
#include <stdio.h>
```

<sup>39</sup>See <https://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html>

```

int main()
{
    int random_value=0x11223344;
    unsigned char array[10];
    int i;
    unsigned char *fakearray=&array[-1];

    for (i=0; i<10; i++)
        array[i]=i;

    printf ("first element %d\n", fakearray[1]);
    printf ("second element %d\n", fakearray[2]);
    printf ("last element %d\n", fakearray[10]);

    printf ("array[-1]=%02X, array[-2]=%02X, array[-3]=%02X, array[-4]=%02X\n",
        array[-1],
        array[-2],
        array[-3],
        array[-4]);
};

```

Листинг 3.120: Неоптимизирующий MSVC 2010

```

1 $SG2751 DB      'first element %d', 0aH, 00H
2 $SG2752 DB      'second element %d', 0aH, 00H
3 $SG2753 DB      'last element %d', 0aH, 00H
4 $SG2754 DB      'array[-1]=%02X, array[-2]=%02X, array[-3]=%02X, array[-4]=%02X', 0aH, 00H
5 DB
6
7 _fakearray$ = -24          ; size = 4
8 _random_value$ = -20      ; size = 4
9 _array$ = -16             ; size = 10
10 _i$ = -4                  ; size = 4
11 _main PROC
12     push    ebp
13     mov     ebp, esp
14     sub     esp, 24
15     mov     DWORD PTR _random_value$[ebp], 287454020 ; 11223344H
16     ; установить fakearray[] на байт раньше перед array[]
17     lea    eax, DWORD PTR _array$[ebp]
18     add    eax, -1 ; eax=eax-1
19     mov     DWORD PTR _fakearray$[ebp], eax
20     mov     DWORD PTR _i$[ebp], 0
21     jmp    SHORT $LN3@main
22     ; заполнить array[] 0..9
23 $LN2@main:
24     mov     ecx, DWORD PTR _i$[ebp]
25     add    ecx, 1
26     mov     DWORD PTR _i$[ebp], ecx
27 $LN3@main:
28     cmp     DWORD PTR _i$[ebp], 10
29     jge    SHORT $LN1@main
30     mov     edx, DWORD PTR _i$[ebp]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

31     mov     al, BYTE PTR _i$[ebp]
32     mov     BYTE PTR _array$[ebp+edx], al
33     jmp     SHORT $LN2@main
34 $LN1@main:
35     mov     ecx, DWORD PTR _fakearray$[ebp]
36     ; ecx=адрес fakearray[0], ecx+1 это fakearray[1] либо array[0]
37     movzx   edx, BYTE PTR [ecx+1]
38     push   edx
39     push   OFFSET $SG2751 ; 'first element %d'
40     call   _printf
41     add     esp, 8
42     mov     eax, DWORD PTR _fakearray$[ebp]
43     ; eax=адрес fakearray[0], eax+2 это fakearray[2] либо array[1]
44     movzx   ecx, BYTE PTR [eax+2]
45     push   ecx
46     push   OFFSET $SG2752 ; 'second element %d'
47     call   _printf
48     add     esp, 8
49     mov     edx, DWORD PTR _fakearray$[ebp]
50     ; edx=адрес fakearray[0], edx+10 это fakearray[10] либо array[9]
51     movzx   eax, BYTE PTR [edx+10]
52     push   eax
53     push   OFFSET $SG2753 ; 'last element %d'
54     call   _printf
55     add     esp, 8
56     ; отнять 4, 3, 2 и 1 от указателя array[0] чтобы найти значения,
    лежащие перед array[]
57     lea    ecx, DWORD PTR _array$[ebp]
58     movzx   edx, BYTE PTR [ecx-4]
59     push   edx
60     lea    eax, DWORD PTR _array$[ebp]
61     movzx   ecx, BYTE PTR [eax-3]
62     push   ecx
63     lea    edx, DWORD PTR _array$[ebp]
64     movzx   eax, BYTE PTR [edx-2]
65     push   eax
66     lea    ecx, DWORD PTR _array$[ebp]
67     movzx   edx, BYTE PTR [ecx-1]
68     push   edx
69     push   OFFSET $SG2754 ;
    'array[-1]=%02X, array[-2]=%02X, array[-3]=%02X, array[-4]=%02X'
70     call   _printf
71     add     esp, 20
72     xor     eax, eax
73     mov     esp, ebp
74     pop     ebp
75     ret     0
76 _main    ENDP

```

Так что у нас тут массив `array[]` из десяти элементов, заполненный байтами `0...9`.

Затем у нас указатель `fakearray[]` указывающий на один байт перед `array[]`. `fakearray[1]` указывает точно на `array[0]`. Но нам все еще любопытно, что

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

же находится перед `array[]`? Мы добавляем `random_value` перед `array[]` и установим её в `0x11223344`. Неоптимизирующий компилятор выделяет переменные в том же порядке, в котором они объявлены, так что да, 32-битная `random_value` находится точно перед массивом.

Запускаем, и:

```
first element 0
second element 1
last element 9
array[-1]=11, array[-2]=22, array[-3]=33, array[-4]=44
```

Фрагмент стека, который мы скопируем из окна стека в OllyDbg (включая комментарии автора):

Листинг 3.121: Неоптимизирующий MSVC 2010

```
CPU Stack
Address  Value
001DFBCC /001DFBD3 ; указатель fakearray
001DFBD0 |11223344 ; random_value
001DFBD4 |03020100 ; 4 байта array[]
001DFBD8 |07060504 ; 4 байта array[]
001DFBDC |00CB0908 ; случайный мусор + 2 последних байта array[]
001DFBE0 |0000000A ; последнее значение i после того как закончился цикл
001DFBE4 |001DFC2C ; сохраненное значение EBP
001DFBE8 \00CB129D ; адрес возврата (RA)
```

Указатель на `fakearray[]` (`0x001DFBD3`) это действительно адрес `array[]` в стеке (`0x001DFBD4`), но минус 1 байт.

Трюк этот все-таки слишком хакерский и сомнительный. Вряд ли кто-то будет его использовать в своем коде, но для демонстрации, он здесь очень уместен.

## 3.21. Больше об указателях

The way C handles pointers, for example, was a brilliant innovation; it solved a lot of problems that we had before in data structuring and made the programs look good afterwards.

Дональд Кнут, интервью (1993)

Для тех, кому все еще трудно понимать указатели в Си/Си++, вот еще примеры. Некоторые из них крайне странные и служат только демонстрационным целям: использовать подобное в production-коде можно только если вы действительно понимаете, что вы делаете.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



### 3.21.1. Работа с адресами вместо указателей

Указатель это просто адрес в памяти. Но почему мы пишем `char* string` вместо чего-нибудь вроде `address string`? Переменная-указатель дополнена типом переменной, на которую указатель указывает. Тогда у компилятора будет возможность находить потенциальные ошибки типизации во время компиляции.

Если быть педантом, типизация данных в языках программирования существует для предотвращения ошибок и самодокументации. Вполне возможно использовать только два типа данных вроде `int` (или `int64_t`) и байт — это те единственные типы, которые доступны для программистов на ассемблере. Но написать что-то большое и практичное на ассемблере, при этом без ошибок, это трудная задача. Любая мелкая опечатка может привести к труднонаходимой ошибке.

Информации о типах нет в скомпилированном коде (и это одна из основных проблем для декомпиляторов), и я могу это продемонстрировать.

Вот как напишет обычный программист на Си/Си++:

```
#include <stdio.h>
#include <stdint.h>

void print_string (char *s)
{
    printf ("(address: 0x%llx)\n", s);
    printf ("%s\n", s);
};

int main()
{
    char *s="Hello, world!";

    print_string (s);
};
```

А вот что могу написать я:

```
#include <stdio.h>
#include <stdint.h>

void print_string (uint64_t address)
{
    printf ("(address: 0x%llx)\n", address);
    puts ((char*)address);
};

int main()
{
    char *s="Hello, world!";

    print_string ((uint64_t)s);
};
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Я использую `uint64_t` потому что я запускаю этот пример на Linux x64. `int` сгодится для 32-битных **OS!**. В начале, указатель на символ (самый первый в строке с приветствием) приводится к `uint64_t`, затем он передается далее. Ф-ция `print_string()` приводит тип переданного значения из `uint64_t` в указатель на символ.

Но вот что интересно, это то что GCC 4.8.4 генерирует идентичный результат на ассемблере для обеих версий:

```
gcc 1.c -S -masm=intel -O3 -fno-inline
```

```
.LC0:
    .string "(address: 0x%llx)\n"
print_string:
    push    rbx
    mov     rdx, rdi
    mov     rbx, rdi
    mov     esi, OFFSET FLAT:.LC0
    mov     edi, 1
    xor     eax, eax
    call   __printf_chk
    mov     rdi, rbx
    pop     rbx
    jmp     puts
.LC1:
    .string "Hello, world!"
main:
    sub     rsp, 8
    mov     edi, OFFSET FLAT:.LC1
    call   print_string
    add     rsp, 8
    ret
```

(Я убрал незначительные директивы GCC.)

Я также пробовал утилиту UNIX `diff` и не нашел разницы вообще.

Продолжим и дальше издеваться над традициями программирования в Си/-Си++. Кто-то может написать так:

```
#include <stdio.h>
#include <stdint.h>

uint8_t load_byte_at_address (uint8_t* address)
{
    return *address;
    //this is also possible: return address[0];
};

void print_string (char *s)
{
    char* current_address=s;
    while (1)
    {
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        char current_char=load_byte_at_address(current_address);
        if (current_char==0)
            break;
        printf ("%c", current_char);
        current_address++;
    };
};

int main()
{
    char *s="Hello, world!";

    print_string (s);
};

```

И это может быть переписано так:

```

#include <stdio.h>
#include <stdint.h>

uint8_t load_byte_at_address (uint64_t address)
{
    return *(uint8_t*)address;
};

void print_string (uint64_t address)
{
    uint64_t current_address=address;
    while (1)
    {
        char current_char=load_byte_at_address(current_address);
        if (current_char==0)
            break;
        printf ("%c", current_char);
        current_address++;
    };
};

int main()
{
    char *s="Hello, world!";

    print_string ((uint64_t)s);
};

```

И тот и другой исходный код преобразуется в одинаковый результат на ассемблере:

```
gcc 1.c -S -masm=intel -O3 -fno-inline
```

```
load_byte_at_address:
    movzx  eax, BYTE PTR [rdi]
    ret

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

print_string:
.LFB15:
    push    rbx
    mov     rbx, rdi
    jmp     .L4
.L7:
    movsx   edi, al
    add     rbx, 1
    call    putchar
.L4:
    mov     rdi, rbx
    call    load_byte_at_address
    test    al, al
    jne     .L7
    pop     rbx
    ret
.LC0:
    .string "Hello, world!"
main:
    sub     rsp, 8
    mov     edi, OFFSET FLAT:.LC0
    call    print_string
    add     rsp, 8
    ret

```

(Здесь я также убрал незначительные директивы GCC.)

Разницы нет: указатели в Си/Си++, в сущности, адреса, но несут в себе также информацию о типе, чтобы предотвратить ошибки во время компиляции. Типы не проверяются во время исполнения, иначе это был бы огромный (и ненужный) прирост времени исполнения.

### 3.21.2. Передача значений как указателей; тэггированные объединения

Вот как можно передавать обычные значения как указатели:

```

#include <stdio.h>
#include <stdint.h>

uint64_t multiply1 (uint64_t a, uint64_t b)
{
    return a*b;
};

uint64_t* multiply2 (uint64_t *a, uint64_t *b)
{
    return (uint64_t*)((uint64_t)a*(uint64_t)b);
};

int main()
{
    printf ("%d\n", multiply1(123, 456));
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
printf ("%d\n", (uint64_t)multiply2((uint64_t*)123, (uint64_t*)456)↵);
};
```

Это работает нормально и GCC 4.8.4 компилирует обе ф-ции multiply1() и multiply2() полностью идентично!

```
multiply1:
    mov    rax, rdi
    imul  rax, rsi
    ret

multiply2:
    mov    rax, rdi
    imul  rax, rsi
    ret
```

Пока вы не разыменовываете указатель (*dereference*) (иными словами, если вы не пытаетесь прочитать данные по адресу в указателе), всё будет работать нормально. Указатель это переменная, которая может содержать что угодно, как и обычная переменная.

Здесь используется инструкция для знакового умножения (IMUL) вместо беззнакового (MUL), об этом читайте больше здесь: ?? (стр. ??).

Кстати, это широко известный хак, называющийся *tagged pointers*. Если коротко, если все ваши указатели указывают на блоки в памяти размером, скажем, 16 байт (или они всегда выровнены по 16-байтной границе), 4 младших бита указателя будут всегда нулевыми, и это пространство может быть как-то использовано. Это очень популярно в компиляторах и интерпретаторах LISP. Они хранят тип ячейки/объекта в неиспользующихся битах, и так можно сэкономить немного памяти. И более того — имея только указатель, можно сразу выяснить тип ячейки/объекта, без дополнительного обращения к памяти. Читайте об этом больше: [Денис Юричев, *Заметки о языке программирования Си/Си++1.3*].

### 3.21.3. Издевательство над указателями в ядре Windows

Секция ресурсов в исполняемых файлах типа PE в Windows это секция, содержащая картинки, иконки, строки, и т. д. Ранние версии Windows позволяли иметь к ним доступ только при помощи идентификаторов, но потом в Microsoft добавили также и способ адресовать ресурсы при помощи строк.

Так что потом стало возможным передать идентификатор или строку в ф-цию [FindResource\(\)](#). Которая декларирована вот так:

```
HRSRC WINAPI FindResource(
    _In_opt_ HMODULE hModule,
    _In_     LPCTSTR lpName,
    _In_     LPCTSTR lpType
);
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

*lpName* и *lpType* имеют тип *char\** или *wchar\**, и когда кто-то всё еще хочет передать идентификатор, нужно использовать макрос [MAKEINTRESOURCE](#), вот так:

```
result = FindResource(..., MAKEINTRESOURCE(1234), ...);
```

Очень интересно то, что всё что делает MAKEINTRESOURCE это приводит целочисленное к указателю. В MSVC 2013, в файле *Microsoft SDKs\Windows\v7.1A\Include\Ks.h*, мы можем найти это:

```
...
#if (!defined( MAKEINTRESOURCE ))
#define MAKEINTRESOURCE( res ) ((ULONG_PTR) (USHORT) res)
#endif
...
```

Звучит безумно. Заглянем внутрь древнего, когда-то утекшего, исходного кода Windows NT4. В *private/windows/base/client/module.c* мы можем найти исходный код *FindResource()*:

```
HRSRC
FindResourceA(
    HMODULE hModule,
    LPCSTR lpName,
    LPCSTR lpType
)
...
{
    NTSTATUS Status;
    ULONG IdPath[ 3 ];
    PVOID p;

    IdPath[ 0 ] = 0;
    IdPath[ 1 ] = 0;
    try {
        if ((IdPath[ 0 ] = BaseDllMapResourceIdA( lpType )) == -1) {
            Status = STATUS_INVALID_PARAMETER;
        }
        else
            if ((IdPath[ 1 ] = BaseDllMapResourceIdA( lpName )) == -1) {
                Status = STATUS_INVALID_PARAMETER;
            }
    }
    ...
}
```

Посмотрим в *BaseDllMapResourceIdA()* в том же исходном файле:

```
ULONG
BaseDllMapResourceIdA(
    LPCSTR lpId
)
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

{
    NTSTATUS Status;
    ULONG Id;
    UNICODE_STRING UnicodeString;
    ANSI_STRING AnsiString;
    PWSTR s;

    try {
        if ((ULONG)lpId & LDR_RESOURCE_ID_NAME_MASK) {
            if (*lpId == '#') {
                Status = RtlCharToInteger( lpId+1, 10, &Id );
                if (!NT_SUCCESS( Status ) || Id & LDR_RESOURCE_ID_NAME_MASK)
            ) {
                    if (NT_SUCCESS( Status )) {
                        Status = STATUS_INVALID_PARAMETER;
                    }
                    BaseSetLastNTErrror( Status );
                    Id = (ULONG)-1;
                }
            }
            else {
                RtlInitAnsiString( &AnsiString, lpId );
                Status = RtlAnsiStringToUnicodeString( &UnicodeString,
                                                       &AnsiString,
                                                       TRUE
                                                       );

                if (!NT_SUCCESS( Status )){
                    BaseSetLastNTErrror( Status );
                    Id = (ULONG)-1;
                }
                else {
                    s = UnicodeString.Buffer;
                    while (*s != UNICODE_NULL) {
                        *s = RtlUppcaseUnicodeChar( *s );
                        s++;
                    }

                    Id = (ULONG)UnicodeString.Buffer;
                }
            }
        }
        else {
            Id = (ULONG)lpId;
        }
    }
    except (EXCEPTION_EXECUTE_HANDLER) {
        BaseSetLastNTErrror( GetExceptionCode() );
        Id = (ULONG)-1;
    }
    return Id;
}

```

К *lpId* применяется операция “И” с *LDR\_RESOURCE\_ID\_NAME\_MASK*.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Маску можно найти в *public/sdk/inc/ntldr.h*:

```
...
#define LDR_RESOURCE_ID_NAME_MASK 0xFFFF0000
...
```

Так что к *lpId* применяется операция “И” с *0xFFFF0000*, и если присутствуют какие-либо биты за младшими 16 битами, выполняется первая часть ф-ции и (*lpId* принимается за адрес строки). Иначе — вторая часть ф-ции (*lpId* принимается за 16-битное значение).

Этот же код можно найти и в Windows 7, в файле *kernel32.dll*:

```
....
.text:0000000078D24510 ;
      __int64 __fastcall BaseDllMapResourceIdA(PCSZ SourceString)
.text:0000000078D24510 BaseDllMapResourceIdA proc near ; CODE XREF:
      FindResourceExA+34
.text:0000000078D24510 ;
      FindResourceExA+4B
.text:0000000078D24510
.text:0000000078D24510 var_38 = qword ptr -38h
.text:0000000078D24510 var_30 = qword ptr -30h
.text:0000000078D24510 var_28 = _UNICODE_STRING ptr -28h
.text:0000000078D24510 DestinationString= _STRING ptr -18h
.text:0000000078D24510 arg_8 = dword ptr 10h
.text:0000000078D24510
.text:0000000078D24510 ; FUNCTION CHUNK AT .text:0000000078D42FB4 SIZE
      000000D5 BYTES
.text:0000000078D24510
.text:0000000078D24510 push rbx
.text:0000000078D24512 sub rsp, 50h
.text:0000000078D24516 cmp rcx, 10000h
.text:0000000078D2451D jnb loc_78D42FB4
.text:0000000078D24523 mov [rsp+58h+var_38], rcx
.text:0000000078D24528 jmp short $+2
.text:0000000078D2452A ;
-----
.text:0000000078D2452A
.text:0000000078D2452A loc_78D2452A: ; CODE XREF:
      BaseDllMapResourceIdA+18
.text:0000000078D2452A ;
      BaseDllMapResourceIdA+1EAD0
.text:0000000078D2452A jmp short $+2
.text:0000000078D2452C ;
-----
.text:0000000078D2452C
.text:0000000078D2452C loc_78D2452C: ;
      CODE XREF: BaseDllMapResourceIdA:loc_78D2452A
.text:0000000078D2452C ;
      BaseDllMapResourceIdA+1EB74
.text:0000000078D2452C mov rax, rcx
.text:0000000078D2452F add rsp, 50h
.text:0000000078D24533 pop rbx
.text:0000000078D24534 retn
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

.text:0000000078D24534 ;
.text:0000000078D24535 align 20h
.text:0000000078D24535 BaseDllMapResourceIdA endp

....

.text:0000000078D42FB4 loc_78D42FB4: ; CODE XREF:
BaseDllMapResourceIdA+D
.text:0000000078D42FB4 cmp byte ptr [rcx], '#'
.text:0000000078D42FB7 jnz short loc_78D43005
.text:0000000078D42FB9 inc rcx
.text:0000000078D42FBC lea r8, [rsp+58h+arg_8]
.text:0000000078D42FC1 mov edx, 0Ah
.text:0000000078D42FC6 call cs:__imp_RtlCharToInteger
.text:0000000078D42FCC mov ecx, [rsp+58h+arg_8]
.text:0000000078D42FD0 mov [rsp+58h+var_38], rcx
.text:0000000078D42FD5 test eax, eax
.text:0000000078D42FD7 js short loc_78D42FE6
.text:0000000078D42FD9 test rcx, 0FFFFFFFFFFFFFF0000h
.text:0000000078D42FE0 jz loc_78D2452A

....

```

Если значение больше чем 0x10000, происходит переход в то место, где обрабатывается строка. Иначе, входное значение *lpId* возвращается как есть. Маска 0xFFFF0000 здесь больше не используется, т.к., это все же 64-битный код, но всё-таки, маска 0xFFFFFFFFFFFF0000 могла бы здесь использоваться.

Внимательный читатель может спросить, что если адрес входной строки будет ниже 0x10000? Этот код полагается на тот факт, что в Windows нет ничего по адресам ниже 0x10000, по крайней мере, в Win32.

Raymond Chen [пишет](#) об этом:

How does MAKEINTRESOURCE work? It just stashes the integer in the bottom 16 bits of a pointer, leaving the upper bits zero. This relies on the convention that the first 64KB of address space is never mapped to valid memory, a convention that is enforced starting in Windows 7.

Коротко говоря, это грязный хак, и наверное не стоит его использовать, если только нет большой необходимости. Вероятно, аргумент ф-ции *FindResource()* в прошлом имел тип *SHORT*, а потом в Microsoft добавили возможность передавать здесь и строки, но старый код также нужно было поддерживать.

Вот мой короткий очищенный пример:

```

#include <stdio.h>
#include <stdint.h>

void f(char* a)
{
    if (((uint64_t)a)>0x10000)

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        printf ("Pointer to string has been passed: %s\n", a);
    else
        printf ("16-bit value has been passed: %d\n", (uint64_t)a);
};

int main()
{
    f("Hello!"); // pass string
    f((char*)1234); // pass 16-bit value
};

```

Работает!

### Издавательство над указателями в ядре Linux

Как было упомянуто среди [комментариев на Hacker News](#), в ядре Linux также есть что-то подобное.

Например, эта ф-ция может возвращать и код ошибки и указатель:

```

struct kernfs_node *kernfs_create_link(struct kernfs_node *parent,
                                       const char *name,
                                       struct kernfs_node *target)
{
    struct kernfs_node *kn;
    int error;

    kn = kernfs_new_node(parent, name, S_IFLNK|S_IRWXUGO, KERNFS_LINK);
    if (!kn)
        return ERR_PTR(-ENOMEM);

    if (kernfs_ns_enabled(parent))
        kn->ns = target->ns;
    kn->symlink.target_kn = target;
    kernfs_get(target); /* ref owned by symlink */

    error = kernfs_add_one(kn);
    if (!error)
        return kn;

    kernfs_put(kn);
    return ERR_PTR(error);
}

```

( <https://github.com/torvalds/linux/blob/fceef393a538134f03b778c5d2519e670269342f/fs/kernfs/symlink.c#L25> )

*ERR\_PTR* это макрос, приводящий целочисленное к указателю:

```

static inline void * __must_check ERR_PTR(long error)
{
    return (void *) error;
}

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

( <https://github.com/torvalds/linux/blob/61d0b5a4b2777dcf5daef245e212b3c1fa8091ca/tools/virtio/linux/err.h> )

Этот же заголовочный файл имеет также макрос, который можно использовать, чтобы отличить код ошибки от указателя:

```
#define IS_ERR_VALUE(x) unlikely((x) >= (unsigned long)-MAX_ERRNO)
```

Это означает, коды ошибок это “указатели” очень близкие к -1, и, будем надеяться, в памяти ядра ничего не находится по адресам вроде 0xFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFE, 0xFFFFFFFFFFFFFFFD, и т. д.

Намного более популярный способ это возвращать *NULL* в случае ошибки и передавать код ошибки через дополнительный аргумент. Авторы ядра Linux так не делают, но все кто пользуется этими ф-циями, должны помнить, что возвращаемый указатель должен быть вначале проверен при помощи *IS\_ERR\_VALUE* перед разыменовыванием.

Например:

```
fman->cam_offset = fman_muram_alloc(fman->muram, fman->cam_size);
if (IS_ERR_VALUE(fman->cam_offset)) {
    dev_err(fman->dev, "%s: MURAM alloc for DMA CAM failed\n",
            __func__);
    return -ENOMEM;
}
```

( <https://github.com/torvalds/linux/blob/aa00edc1287a693eadc7bc67a3d73555d969b35d/drivers/net/ethernet/freescale/fman/fman.c#L826> )

### Издавательство над указателями в пользовательской среде UNIX

Ф-ция *mmap()* возвращает -1 в случае ошибки (или *MAP\_FAILED*, что равно -1). Некоторые люди говорят, что в некоторых случаях, *mmap()* может подключить память по нулевому адресу, так что использовать 0 или *NULL* как код ошибки нельзя.

## 3.21.4. Нулевые указатели

### Ошибка “Null pointer assignment” во времена MS-DOS

Читатели постарше могут помнить очень странную ошибку эпохи MS-DOS: “Null pointer assignment”. Что она означает?

В \*NIX и Windows нельзя записывать в память по нулевому адресу, но это было возможно в MS-DOS, из-за отсутствия защиты памяти как таковой.

Так что я могу найти древний Turbo C++ 3.0 (позже он был переименован в C++) из начала 1990-х и попытаться скомпилировать это:

```
#include <stdio.h>

int main()
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
{
    int *ptr=NULL;
    *ptr=1234;
    printf ("Now let's read at NULL\n");
    printf ("%d\n", *ptr);
};
```

Трудно поверить, но это работает, но с ошибкой при выходе:

#### Листинг 3.122: Древний Turbo C++ 3.0

```
C:\TC30\BIN\1
Now let's read at NULL
1234
Null pointer assignment
C:\TC30\BIN>_
```

Посмотрим внутри исходного кода **CRT!** компилятора Borland C++ 3.1, файл *c0.asm*:

```
; _checknull()    check for null pointer zapping copyright message
...
; Check for null pointers before exit

__checknull      PROC    DIST
                  PUBLIC  __checknull

IF               LDATA EQ false
  IFNDEF        __TINY__
                  push    si
                  push    di
                  mov     es, cs:DGROUP@@
                  xor     ax, ax
                  mov     si, ax
                  mov     cx, lgth_CopyRight
ComputeChecksum  label near
                  add     al, es:[si]
                  adc     ah, 0
                  inc     si
                  loop   ComputeChecksum
                  sub     ax, CheckSum
                  jz     @@SumOK
                  mov     cx, lgth_NullCheck
                  mov     dx, offset DGROUP: NullCheck
                  call    ErrorDisplay
@@SumOK:         pop     di
                  pop     si
                  ENDIF
ENDIF

_DATA           SEGMENT
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; Magic symbol used by the debug info to locate the data segment
public DATASEG@
DATASEG@      label byte

; The Copyright string must NOT be moved or changed without
; changing the null pointer check logic

Copyright     db      4 dup(0)
              db      'Borland C++ - Copyright 1991 Borland Intl.',0
lgth_CopyRight equ   $ - Copyright

IF          LDATA EQ false
IFNDEF     __TINY__
Checksum   equ      00D5Ch
NullCheck  db       'Null pointer assignment', 13, 10
lgth_NullCheck equ   $ - NullCheck
ENDIF
ENDIF
...

```

Модель памяти в MS-DOS крайне странная (?? (стр. ??)), и, вероятно, её и не нужно изучать, если только вы не фанат ретрокомпьютинга или ретрогейминга. Одну только вещь можно держать в памяти, это то, что сегмент памяти (включая сегмент данных) в MS-DOS это место где хранится код или данные, но в отличие от “серьезных” **OS!**, он начинается с нулевого адреса.

И в Borland C++ **CRT!**, сегмент данных начинается с 4-х нулевых байт и строки копирайта “Borland C++ - Copyright 1991 Borland Intl.”. Целостность 4-х нулевых байт и текстовой строки проверяется в конце, и если что-то нарушено, выводится сообщение об ошибке.

Но зачем? Запись по нулевому указателю это распространенная ошибка в Си/Си++, и если вы делаете это в \*NIX или Windows, ваше приложение упадет. В MS-DOS нет защиты памяти, так что это приходится проверять в **CRT!** во время выхода, пост-фактум. Если вы видите это сообщение, значит ваша программа в каком-то месте что-то записала по нулевому адресу.

Наша программа это сделала. И вот почему число 1234 было прочитано корректно: потому что оно было записано на месте первых 4-х байт. Контрольная сумма во время выхода неверна (потому что наше число там осталось), так что сообщение было выведено.

Прав ли я? Я переписал программу для проверки моих предположений:

```

#include <stdio.h>

int main()
{
    int *ptr=NULL;
    *ptr=1234;
    printf ("Now let's read at NULL\n");
    printf ("%d\n", *ptr);
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
*ptr=0; // psst, cover our tracks!
};
```

Программа исполняется без ошибки во время выхода.

Хотя и метод предупреждать о записи по нулевому указателю имел смысл в MS-DOS, вероятно, это всё может использоваться и сегодня, на маломощных **MCU!** без защиты памяти и/или **MMU!**<sup>40</sup>.

### Почему кому-то может понадобиться писать по нулевому адресу?

Но почему трезвомыслящему программисту может понадобиться записывать что-то по нулевому адресу? Это может быть сделано случайно, например, указатель должен быть инициализирован и указывать на только что выделенный блок в памяти, а затем должен быть передан в какую-то ф-цию, возвращающую данные через указатель.

```
int *ptr=NULL;

... мы забыли выделить память и инициализировать ptr

strcpy (ptr, buf); // strcpy() завершает работу молча, потому что в MS-DOS
    НЕТ ЗАЩИТЫ ПАМЯТИ
```

И даже хуже:

```
int *ptr=malloc(1000);

... мы забыли проверить, действительно ли память была выделена: это же MS-↵
    ↵ DOS и у тогдашних компьютеров было мало памяти,
... и нехватка памяти была обычной ситуацией.
... если malloc() вернул NULL, тогда ptr будет тоже NULL.

strcpy (ptr, buf); // strcpy() завершает работу молча, потому что в MS-DOS
    НЕТ ЗАЩИТЫ ПАМЯТИ
```

### Писать по нулевому адресу намеренно

Вот пример из `dmalloc`<sup>41</sup>, портативный (переносимый) способ сгенерировать core dump, в отсутствие иных способов:

#### 3.4 Generating a Core File on Errors

```
=====

If the `error-abort' debug token has been enabled, when the library
detects any problems with the heap memory, it will immediately attempt
to dump a core file. *Note Debug Tokens::. Core files are a complete
copy of the program and it's state and can be used by a debugger to see
specifically what is going on when the error occurred. *Note Using
With a Debugger::. By default, the low, medium, and high arguments to
```

<sup>40</sup> **MMU!**

<sup>41</sup> <http://dmalloc.com/>

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

the library utility enable the `error-abort` token. You can disable this feature by entering `dmalloc -m error-abort` (-m for minus) to remove the `error-abort` token and your program will just log errors and continue. You can also use the `error-dump` token which tries to dump core when it sees an error but still continue running. \*Note Debug Tokens::.

When a program dumps core, the system writes the program and all of its memory to a file on disk usually named `core`. If your program is called `foo` then your system may dump core as `foo.core`. If you are not getting a `core` file, make sure that your program has not changed to a new directory meaning that it may have written the core file in a different location. Also insure that your program has write privileges over the directory that it is in otherwise it will not be able to dump a core file. Core dumps are often security problems since they contain all program memory so systems often block their being produced. You will want to check your user and system's core dump size ulimit settings.

The library by default uses the `abort` function to dump core which may or may not work depending on your operating system. If the following program does not dump core then this may be the problem. See `KILL\_PROCESS` definition in `settings.dist`.

```
main()
{
    abort();
}
```

If `abort` does work then you may want to try the following setting in `settings.dist`. This code tries to generate a segmentation fault by dereferencing a `NULL` pointer.

```
#define KILL_PROCESS    { int *_int_p = 0L; *_int_p = 1; }
```

## NULL в Си/Си++

NULL в C/C++ это просто макрос, который часто определяют так:

```
#define NULL ((void*)0)
```

( [libio.h file](#) )

*void\** это тип данных, отражающий тот факт, что это указатель, но на значение неизвестного типа (*void*).

NULL обычно используется чтобы показать отсутствие объекта. Например, у вас есть односвязный список, и каждый узел имеет значение (или указатель на значение) и указатель вроде *next*. Чтобы показать, что следующего узла нет, в поле *next* записывается 0. (Остальные решения просто хуже.) Вероятно, вы можете использовать какую-то крайне экзотическую среду, где можно выделить память по нулевому адресу. Как вы будете показывать отсутствие

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

следующего узла? Какой-нибудь *magic number*? Может быть -1? Или дополнительным битом?

В Википедии мы можем найти это:

In fact, quite contrary to the zero page's original preferential use, some modern operating systems such as FreeBSD, Linux and Microsoft Windows[2] actually make the zero page inaccessible to trap uses of NULL pointers.

( [https://en.wikipedia.org/wiki/Zero\\_page](https://en.wikipedia.org/wiki/Zero_page) )

### Нулевой указатель на ф-цию

Можно вызывать ф-ции по их адресу. Например, я компилирую это при помощи MSVC 2010 и запускаю в Windows 7:

```
#include <windows.h>
#include <stdio.h>

int main()
{
    printf ("0x%x\n", &MessageBoxA);
};
```

Результат `0x7578feae`, и он не меняется и после того, как я запустил это несколько раз, потому что `user32.dll` (где находится ф-ция `MessageBoxA`) всегда загружается по одному и тому же адресу. И потому что **ASLR!**<sup>42</sup> не включено (тогда результат был бы всё время разным).

Вызовем ф-цию `MessageBoxA()` по адресу:

```
#include <windows.h>
#include <stdio.h>

typedef int (*msgboxtype)(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption,
    ↵ UINT uType);

int main()
{
    msgboxtype msgboxaddr=0x7578feae;

    // заставить загрузиться DLL в память процесса,
    // т.к., наш код не использует никакую ф-цию из user32.dll,
    // и DLL не импортируется
    LoadLibrary ("user32.dll");

    msgboxaddr(NULL, "Hello, world!", "hello", MB_OK);
};
```

<sup>42</sup>**ASLR!**



Странно выглядит, но работает в Windows 7 x86.

Это часто используется в шелл-кодах, потому что оттуда трудно вызывать ф-ции из DLL по их именам. А **ASLR!** это контрмера.

И вот теперь что по-настоящему странно выглядит, некоторые программисты на Си для встраиваемых (embedded) систем, могут быть знакомы с таким кодом:

```
int reset()
{
    void (*foo)(void) = 0;
    foo();
};
```

Кому понадобится вызывать ф-цию по адресу 0? Это портбельный способ перейти на нулевой адрес. Множество маломощных микроконтроллеров не имеют защиты памяти или **MMU!**, и после сброса, они просто начинают исполнять код по нулевому адресу, где может быть записан инициализирующий код. Так что переход по нулевому адресу это способ сброса. Можно использовать и inline-ассемблер, но если это неудобно, тогда можно использовать этот портбельный метод.

Это даже корректно компилируется при помощи GCC 4.8.4 на Linux x64:

```
reset:
    sub    rsp, 8
    xor    eax, eax
    call   rax
    add    rsp, 8
    ret
```

То обстоятельство, что указатель стека сдвинут, это не проблема: инициализирующий код в микроконтроллерах обычно полностью игнорирует состояние регистров и памяти и загружает всё “с чистого листа”.

И конечно, этот код упадет в \*NIX или Windows, из-за защиты памяти, и даже если бы её не было, по нулевому адресу нет никакого кода.

В GCC даже есть нестандартное расширение, позволяющее перейти по определенному адресу, вместо того чтобы вызывать ф-цию: <http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html>.

### 3.21.5. Массив как аргумент функции

Кто-то может спросить, какая разница между объявлением аргумента ф-ции как массива и как указателя?

Как видно, разницы вообще нет:

```
void write_something1(int a[16])
{
    a[5]=0;
};
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

void write_something2(int *a)
{
    a[5]=0;
};

int f()
{
    int a[16];
    write_something1(a);
    write_something2(a);
};

```

Оптимизирующий GCC 4.8.4:

```

write_something1:
    mov     DWORD PTR [rdi+20], 0
    ret

write_something2:
    mov     DWORD PTR [rdi+20], 0
    ret

```

Но вы можете объявлять массив вместо указателя для самодокументации, если размер массива известен заранее и определен. И может быть, какой-нибудь инструмент для статического анализа выявит возможное переполнение буфера. Или такие инструменты есть уже сегодня?

Некоторые люди, включая Линуса Торвальдса, критикуют эту возможность Си/Си++: <https://lkml.org/lkml/2015/9/3/428>.

В стандарте C99 имеется также ключевое слово *static* [ISO/IEC 9899:TC3 (C C99 standard), (2007) 6.7.5.3]:

If the keyword *static* also appears within the [ and ] of the array type derivation, then for each call to the function, the value of the corresponding actual argument shall provide access to the first element of an array with at least as many elements as specified by the size expression.

### 3.21.6. Указатель на функцию

Имя ф-ции в Си/Си++ без скобок, как "printf" это указатель на ф-цию типа *void (\*)()*. Попробуем прочитать содержимое ф-ции и пропатчить его:

```

#include <memory.h>
#include <stdio.h>

void print_something ()
{
    printf ("we are in %s()\n", __FUNCTION__);
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
};

int main()
{
    print_something();
    printf ("first 3 bytes: %x %x %x...\n",
           *(unsigned char*)print_something,
           *((unsigned char*)print_something+1),
           *((unsigned char*)print_something+2));

    *(unsigned char*)print_something=0xC3; // RET's opcode
    printf ("going to call patched print_something():\n");
    print_something();
    printf ("it must exit at this point\n");
};
```

При запуске видно что первые 3 байта ф-ции это 55 89 e5. Действительно, это опкоды инструкций PUSH EBP и MOV EBP, ESP (это опкоды x86). Но потом процесс падает, потому что секция *text* доступна только для чтения.

Мы можем перекомпилировать наш пример и сделать так, чтобы секция *text* была доступна для записи <sup>43</sup>:

```
gcc --static -g -Wl,--omagic -o example example.c
```

Это работает!

```
we are in print_something()
first 3 bytes: 55 89 e5...
going to call patched print_something():
it must exit at this point
```

### 3.21.7. Указатель на функцию: защита от копирования

Взломщик может найти ф-цию, проверяющую защиту и возвращать *true* или *false*. Он(а) может вписать там XOR EAX, EAX / RETN или MOV EAX, 1 / RETN.

Может ли проверить целостность ф-ции? Оказывается, сделать это легко.

Судя по objdump, первые 3 байта ф-ции *check\_protection()* это 0x55 0x89 0xE5 (учитывая, что это неоптимизирующий GCC):

```
#include <stdlib.h>
#include <stdio.h>

int check_protection()
{
    // do something
    return 0;
    // or return 1;
```

<sup>43</sup><http://stackoverflow.com/questions/27581279/make-text-segment-writable-elf>

```

};

int main()
{
    if (check_protection()==0)
    {
        printf ("no protection installed\n");
        exit(0);
    };

    // ...and then, at some very important point...
    if (*((unsigned char*)check_protection)+0) != 0x55)
    {
        printf ("1st byte has been altered\n");
        // do something mean, add watermark, etc
    };
    if (*((unsigned char*)check_protection)+1) != 0x89)
    {
        printf ("2nd byte has been altered\n");
        // do something mean, add watermark, etc
    };
    if (*((unsigned char*)check_protection)+2) != 0xe5)
    {
        printf ("3rd byte has been altered\n");
        // do something mean, add watermark, etc
    };
};

```

```

0000054d <check_protection>:
54d: 55                push   %ebp
54e: 89 e5            mov    %esp,%ebp
550: e8 b7 00 00 00   call   60c <__x86.get_pc_thunk.ax>
555: 05 7f 1a 00 00   add   $0x1a7f,%eax
55a: b8 00 00 00 00   mov   $0x0,%eax
55f: 5d              pop   %ebp
560: c3              ret

```

Если кто-то пропатчит начало ф-ции `check_protection()`, ваша программа может совершить что-то подлое, например, внезапно закончить работу. Чтобы разобраться с таким трюком, взломщик может установить брякпоинт на чтение памяти, по адресу начала ф-ции. (В [tracer](#)-е для этого есть опция `VRMx`.)

### 3.21.8. Указатель на ф-цию: частая ошибка (или опечатка)

Печально известная ошибка/опечатка:

```

int expired()
{
    // check license key, current date/time, etc
};

int main()

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

{
    if (expired) // must be expired() here
    {
        print ("expired\n");
        exit(0);
    }
    else
    {
        // do something
    };
};

```

Т.к. имя ф-ции само по себе трактуется как указатель на ф-цию, или её адрес, выражение `if(function_name)` работает как `if(true)`.

К сожалению, компилятор с Си/Си++ не выдает предупреждение об этом.

### 3.21.9. Указатель как идентификатор объекта

В ассемблере и Си нет возможностей **OOP!**, но там вполне можно писать код в стиле **OOP!** (просто относитесь к структуре, как к объекту).

Интересно что, иногда, указатель на объект (или его адрес) называется идентификатором (в смысле сокрытия данных/инкапсуляции).

Например, `LoadLibrary()`, судя по **MSDN!**<sup>44</sup>, возвращает “handle” модуля<sup>45</sup>. Затем вы передаете этот “handle” в другую ф-цию вроде `GetProcAddress()`. Но на самом деле, `LoadLibrary()` возвращает указатель на DLL-файл загруженный (*mapped*) в памяти<sup>46</sup>. Вы можете прочитать два байта по адресу возвращенному `LoadLibrary()`, и это будет “MZ” (первые два байта любого файла типа .EXE/.DLL в Windows).

Очевидно, Microsoft “скрывает” этот факт для обеспечения лучшей совместимости в будущем. Также, типы данных `HMODULE` и `HINSTANCE` имели другой смысл в 16-битной Windows.

Возможно, это причина, почему `printf()` имеет модификатор “%p”, который используется для вывода указателей (32-битные целочисленные на 32-битных архитектурах, 64-битные на 64-битных, и т. д.) в шестнадцатеричной форме. Адрес структуры сохраненный в отладочном протоколе может помочь в поисках такого же в том же протоколе.

Вот например из исходного кода SQLite:

```

...

struct Pager {
    sqlite3_vfs *pVfs;          /* OS functions to use for IO */

```

<sup>44</sup>**MSDN!**

<sup>45</sup>[https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms684175\(v=vs.85\).aspx](https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms684175(v=vs.85).aspx)

<sup>46</sup><https://blogs.msdn.microsoft.com/oldnewthing/20041025-00/?p=37483>

```

u8 exclusiveMode;          /* Boolean. True if locking_mode==EXCLUSIVE ↵
    ↵ */
u8 journalMode;           /* One of the PAGER_JOURNALMODE_* values */
u8 useJournal;            /* Use a rollback journal on this file */
u8 noSync;                /* Do not sync the journal if true */

....

static int pagerLockDb(Pager *pPager, int eLock){
    int rc = SQLITE_OK;

    assert( eLock==SHARED_LOCK || eLock==RESERVED_LOCK || eLock==↵
    ↵ EXCLUSIVE_LOCK );
    if( pPager->eLock<eLock || pPager->eLock==UNKNOWN_LOCK ){
        rc = sqlite3OsLock(pPager->fd, eLock);
        if( rc==SQLITE_OK && (pPager->eLock!=UNKNOWN_LOCK||eLock==↵
        ↵ EXCLUSIVE_LOCK) ){
            pPager->eLock = (u8)eLock;
            IOTRACE("LOCK %p %d\n", pPager, eLock)
        }
    }
    return rc;
}

...

PAGER_INCR(sqlite3_pager_readdb_count);
PAGER_INCR(pPager->nRead);
IOTRACE(("PGIN %p %d\n", pPager, pgno));
PAGERTRACE(("FETCH %d page %d hash(%08x)\n",
            PAGERID(pPager), pgno, pager_pagehash(pPg)));

...

```

## 3.22. Оптимизации циклов

### 3.22.1. Странная оптимизация циклов

Это самая простая (из всех возможных) реализация memcpy():

```

void memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
    size_t i;
    for (i=0; i<cnt; i++)
        dst[i]=src[i];
};

```

Как минимум MSVC 6.0 из конца 90-х вплоть до MSVC 2013 может выдавать вот такой странный код (этот листинг создан MSVC 2013 x86):

```

_dst$ = 8          ; size = 4

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

_src$ = 12          ; size = 4
_cnt$ = 16         ; size = 4
_memcpy PROC
    mov     edx, DWORD PTR _cnt$[esp-4]
    test   edx, edx
    je     SHORT $LN1@f
    mov     eax, DWORD PTR _dst$[esp-4]
    push   esi
    mov     esi, DWORD PTR _src$[esp]
    sub    esi, eax
; ESI=src-dst, т.е., разница указателей
$LL8@f:
    mov     cl, BYTE PTR [esi+eax] ; загрузить байт на "esi+dst" или на
"src-dst+dst" в начале, или просто на "src"
    lea    eax, DWORD PTR [eax+1] ; dst++
    mov     BYTE PTR [eax-1], cl  ; сохранить байт на "(dst++)--", или
просто на "dst" в начале
    dec    edx                    ; декремент счетчика, пока не закончим
    jne    SHORT $LL8@f
    pop    esi
$LN1@f:
    ret    0
_memcpy ENDP

```

Это всё странно, потому что как люди работают с двумя указателями? Они сохраняют два адреса в двух регистрах или двух ячейках памяти. Компилятор MSVC в данном случае сохраняет два указателя как один указатель (*скользящий dst* в EAX) и разницу между указателями *src* и *dst* (она остается неизменной во время исполнения цикла, в ESI). (Кстати, это тот редкий случай, когда можно использовать тип `ptrdiff_t`.) Когда нужно загрузить байт из *src*, он загружается на *diff + скользящий dst* и сохраняет байт просто на *скользящем dst*.

Должно быть это какой-то трюк для оптимизации. Но я переписал эту ф-цию так:

```

_f2 PROC
    mov     edx, DWORD PTR _cnt$[esp-4]
    test   edx, edx
    je     SHORT $LN1@f
    mov     eax, DWORD PTR _dst$[esp-4]
    push   esi
    mov     esi, DWORD PTR _src$[esp]
; eax=dst; esi=src
$LL8@f:
    mov     cl, BYTE PTR [esi+edx]
    mov     BYTE PTR [eax+edx], cl
    dec    edx
    jne    SHORT $LL8@f
    pop    esi
$LN1@f:
    ret    0
_f2 ENDP

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

...и она работает также быстро как и *соптимизированная* версия на моем Intel Xeon E31220 @ 3.10GHz. Может быть, эта оптимизация предназначалась для более старых x86-процессоров 90-х, т.к., этот трюк использует как минимум древний MS VC 6.0?

Есть идеи?

Hex-Rays 2.2 не распознает такие шаблонные фрагменты кода (будем надеяться, это временно?):

```
void __cdecl f1(char *dst, char *src, size_t size)
{
    size_t counter; // edx@1
    char *sliding_dst; // eax@2
    char tmp; // cl@3

    counter = size;
    if ( size )
    {
        sliding_dst = dst;
        do
        {
            tmp = (sliding_dst++)[src - dst]; // разница (src-dst)
            вычисляется один раз, перед телом цикла
            *(sliding_dst - 1) = tmp;
            --counter;
        }
        while ( counter );
    }
}
```

Тем не менее, этот трюк часто используется в MSVC (и не только в самодельных ф-циях *тетсру()*), но также и во многих циклах, работающих с двумя или более массивами), так что для реверс-инжиниров стоит помнить об этом.

### 3.22.2. Еще одна оптимизация циклов

Если вы обрабатываете все элементы некоторого массива, который находится в глобальной памяти, компилятор может оптимизировать это. Например, вычисляем сумму всех элементов массива из 128-и *int*-ов:

```
#include <stdio.h>

int a[128];

int sum_of_a()
{
    int rt=0;

    for (int i=0; i<128; i++)
        rt=rt+a[i];

    return rt;
};
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

int main()
{
    // инициализация
    for (int i=0; i<128; i++)
        a[i]=i;

    // вычисляем сумму
    printf ("%d\n", sum_of_a());
};

```

Оптимизирующий GCC 5.3.1 (x86) может сделать так (**IDA!**):

```

.text:080484B0 sum_of_a      proc near
.text:080484B0             mov     edx, offset a
.text:080484B5             xor     eax, eax
.text:080484B7             mov     esi, esi
.text:080484B9             lea    edi, [edi+0]
.text:080484C0
.text:080484C0 loc_80484C0:          ; CODE XREF: sum_of_a+1B
.text:080484C0             add     eax, [edx]
.text:080484C2             add     edx, 4
.text:080484C5             cmp     edx, offset ↗
    ↙ __libc_start_main@@GLIBC_2_0
.text:080484CB             jnz    short loc_80484C0
.text:080484CD             rep retn
.text:080484CD sum_of_a      endp
.text:080484CD

...

.bss:0804A040             public a
.bss:0804A040 a                dd 80h dup(?) ; DATA XREF: main:loc_8048338
.bss:0804A040                                     ; main+19
.bss:0804A040 _bss          ends
.bss:0804A040
extern:0804A240 ;
=====
extern:0804A240
extern:0804A240 ; Segment type: Externs
extern:0804A240 ; extern
extern:0804A240             extrn  __libc_start_main@@GLIBC_2_0:near
extern:0804A240                                     ; DATA XREF: main+25
extern:0804A240                                     ; main+5D
extern:0804A244             extrn  __printf_chk@@GLIBC_2_3_4:near
extern:0804A248             extrn  __libc_start_main:near
extern:0804A248                                     ; CODE XREF: __libc_start_main
extern:0804A248                                     ; DATA XREF: .got.plt:off_804A00C

```

И что же такое `__libc_start_main@@GLIBC_2_0` на `0x080484C5`? Это метка, находящаяся сразу за концом массива `a[]`. Эта ф-ция может быть переписана так:

```
int sum_of_a_v2()
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

{
    int *tmp=a;
    int rt=0;

    do
    {
        rt=rt+(*tmp);
        tmp++;
    }
    while (tmp<(a+128));

    return rt;
};

```

Первая версия имеет счетчик *i*, и адрес каждого элемента массива вычисляется на каждой итерации. Вторая версия более оптимизирована: указатель на каждый элемент массива всегда готов, и продвигается на 4 байта вперед на каждой итерации. Как проверить, закончился ли цикл? Просто сравните указатель с адресом сразу за концом массива, который, как случилось в нашем случае, это просто адрес импортируемой из Glibc 2.0 ф-ции `__libc_start_main()`. Такой код иногда сбивает с толку, и это очень популярный оптимизационный трюк, поэтому я сделал этот пример.

Моя вторая версия очень близка к тому, что сделал GCC, и когда я компилирую её, код почти такой как и в первой версии, но две первых инструкции поменены местами:

```

.text:080484D0      public sum_of_a_v2
.text:080484D0 sum_of_a_v2      proc near
.text:080484D0          xor     eax, eax
.text:080484D2          mov     edx, offset a
.text:080484D7          mov     esi, esi
.text:080484D9          lea    edi, [edi+0]
.text:080484E0
.text:080484E0 loc_80484E0:      ; CODE XREF: sum_of_a_v2+1B
.text:080484E0          add     eax, [edx]
.text:080484E2          add     edx, 4
.text:080484E5          cmp     edx, offset ↵
    ↵ __libc_start_main@@GLIBC_2_0
.text:080484EB          jnz    short loc_80484E0
.text:080484ED          rep retn
.text:080484ED sum_of_a_v2      endp

```

Надо сказать, эта оптимизация возможна если компилятор, во время компиляции, может рассчитать адрес за концом массива. Это случается если массив глобальный и его размер фиксирован.

Хотя, если адрес массива не известен во время компиляции, но его размер фиксирован, адрес метки за концом массива можно вычислить в начале цикла.

## 3.23. Еще о структурах

### 3.23.1. Иногда вместо массива можно использовать структуру в Си

#### Арифметическое среднее

```
#include <stdio.h>

int mean(int *a, int len)
{
    int sum=0;
    for (int i=0; i<len; i++)
        sum=sum+a[i];
    return sum/len;
};

struct five_ints
{
    int a0;
    int a1;
    int a2;
    int a3;
    int a4;
};

int main()
{
    struct five_ints a;
    a.a0=123;
    a.a1=456;
    a.a2=789;
    a.a3=10;
    a.a4=100;
    printf ("%d\n", mean(&a, 5));
    // test:
    https://www.wolframalpha.com/input/?i=mean\(123,456,789,10,100\)
};
```

Это работает: ф-ция *mean()* никогда не будет читать за концом структуры *five\_ints*, потому что передано 5, означая, что только 5 целочисленных значений будет прочитано.

#### Сохраняем строку в структуре

```
#include <stdio.h>

struct five_chars
{
    char a0;
    char a1;
    char a2;
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        char a3;
        char a4;
} __attribute__((aligned (1),packed));

int main()
{
    struct five_chars a;
    a.a0='h';
    a.a1='i';
    a.a2='!';
    a.a3='\n';
    a.a4=0;
    printf (&a); // это печатает "hi!"
};

```

Нужно использовать атрибут *((aligned (1),packed))* потому что иначе, каждое поле структуры будет выровнено по 4-байтной или 8-байтной границе.

### Итог

Это просто еще один пример, как структуры и массивы сохраняются в памяти. Вероятно, ни один программист в трезвом уме не будет делать так, как в этом примере, за исключением, может быть, какого-то очень специального хака. Или может быть в случае обфускации исходных текстов?

### 3.23.2. Безразмерный массив в структуре Си

В некоторых win32-структурах мы можем найти такие, где последнее поле определено как массив из одного элемента:

```

typedef struct _SYMBOL_INFO {
    ULONG    SizeOfStruct;
    ULONG    TypeIndex;

    ...

    ULONG    MaxNameLen;
    TCHAR    Name[1];
} SYMBOL_INFO, *PSYMBOL_INFO;

```

( [https://msdn.microsoft.com/en-us/library/windows/desktop/ms680686\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms680686(v=vs.85).aspx) )

Это хак, в том смысле, что последнее поле это массив неизвестной длины, его размер будет вычислен во время выделения памяти под структуру.

Вот почему: поле *Name* может быть коротким, так зачем же тогда определять константу вроде *MAX\_NAME*, которая может быть 128, 256, или даже больше?

Почему вместо этого не использовать указатель? Но тогда придется выделять два блока: один под структуру и второй под строку. Это может быть медленнее

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

и может требовать больше затрат на память. Также, вам нужно разыменовывать указатель (т.е., читать адрес строки из структуры) — не очень большая проблема, но некоторые люди могут сказать вам, что это дополнительные расходы.

Это также известно как *struct hack*: <http://c-faq.com/struct/structhack.html>.

Например:

```
#include <stdio.h>

struct st
{
    int a;
    int b;
    char s[];
};

void f (struct st *s)
{
    printf ("%d %d %s\n", s->a, s->b, s->s);
    // f() не может заменить s[] большей строкой - длина выделенного
    // блока неизвестна на этом этапе
};

int main()
{
#define STRING "Hello!"
    struct st *s=malloc(sizeof(struct st)+strlen(STRING)+1); // включая
    // терминирующий ноль.
    s->a=1;
    s->b=2;
    strcpy (s->s, STRING);
    f(s);
};
```

Если коротко, это работает, потому что в Си нет проверок границ массивов. К любому массиву относятся так, будто он бесконечный.

Проблема: после выделения, полный размер выделенного блока для структуры неизвестен (хотя известен менеджеру памяти), так что вы не можете заменить строку большей строкой. Но вы бы смогли делать это, если бы поле было определено как что-то вроде `s[MAX_NAME]`.

Другими словами, вы имеете структуру плюс массив (или строку) спаянных вместе в одном выделенном блоке памяти. Другая проблема еще в том, что вы не можете объявить два таких массива в одной структуре, или объявить еще одно поле после такого массива.

Более старые компиляторы требуют объявить массив хотя бы с одним элементом: `s[1]`, более новые позволяют определять его как массив с переменной длиной: `s[]`. В стандарте C99 это также называется *flexible array member*.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Читайте об этом больше в документации GCC<sup>47</sup>, в документации MSDN<sup>48</sup>.

Деннис Ритчи (один из создателей Си) называет этот трюк «unwarranted chumminess with the C implementation» (вероятно, подтверждая хакерскую природу трюка).

Вам это может нравиться, или нет, вы можете использовать это или нет: но это еще одна демонстрация того, как структуры располагаются в памяти, вот почему я написал об этом.

### 3.23.3. Версия структуры в Си

Многие программисты под Windows видели это в MSDN:

```
SizeOfStruct
    The size of the structure, in bytes. This member must be set to sizeof(↵
    ↵ SYMBOL_INFO).
```

( [https://msdn.microsoft.com/en-us/library/windows/desktop/ms680686\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms680686(v=vs.85).aspx) )

Некоторые структуры вроде *SYMBOL\_INFO* действительно начинаются с такого поля. Почему? Это что-то вроде версии структуры.

Представьте, что у вас есть ф-ция, рисующая круги. Она берет один аргумент - указатель на структуру с двумя полями: X, Y и радиус. И затем цветные дисплеи наводнили рынок, где-то в 80-х. И вы хотите добавить аргумент *цвет* в ф-цию. Но, скажем так, вы не можете добавить еще один аргумент в нее (множество ПО используют ваше **API!**<sup>49</sup> и его нельзя перекомпилировать). И если какое-то старое ПО использует ваше **API!** с цветным дисплеем, пусть ваша ф-ция рисует круг в цветах по умолчанию (черный и белый).

Позже вы добавляете еще одну возможность: круг может быть покрашен, и можно выбирать тип заливки.

Вот одно из решений проблемы:

```
#include <stdio.h>

struct ver1
{
    size_t SizeOfStruct;
    int coord_X;
    int coord_Y;
    int radius;
};

struct ver2
{
    size_t SizeOfStruct;
    int coord_X;
    int coord_Y;
```

<sup>47</sup><https://gcc.gnu.org/onlinedocs/gcc/Zero-Length.html>

<sup>48</sup><https://msdn.microsoft.com/en-us/library/b6fae073.aspx>

<sup>49</sup>**API!**

```

        int radius;
        int color;
};

struct ver3
{
    size_t SizeOfStruct;
    int coord_X;
    int coord_Y;
    int radius;
    int color;
    int fill_brush_type; // 0 - не заливать круг
};

void draw_circle(struct ver3 *s) // здесь используется самая последняя версия
    структуры
{
    // мы полагаем что в SizeOfStruct всегда присутствуют поля coord_X и
    coord_Y
    printf ("Собираемся рисовать круг на %d:%d\n", s->coord_X, s->
    ↵ coord_Y);

    if (s->SizeOfStruct>=sizeof(int)*5)
    {
        // это минимум ver2, поле цвета присутствует
        printf ("Собираемся установить цвет %d\n", s->color);
    }

    if (s->SizeOfStruct>=sizeof(int)*6)
    {
        // это минимум ver3, присутствует поле с типом заливки
        printf ("Мы собираемся залить его используя тип заливки %d\
    ↵ n", s->fill_brush_type);
    }
};

// раннее ПО
void call_as_ver1()
{
    struct ver1 s;
    s.SizeOfStruct=sizeof(s);
    s.coord_X=123;
    s.coord_Y=456;
    s.radius=10;
    printf ("** %s()\n", __FUNCTION__);
    draw_circle(&s);
};

// следующая версия
void call_as_ver2()
{
    struct ver2 s;
    s.SizeOfStruct=sizeof(s);
    s.coord_X=123;

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        s.coord_Y=456;
        s.radius=10;
        s.color=1;
        printf ("** %s()\n", __FUNCTION__);
        draw_circle(&s);
};

// самая поздняя, наиболее расширенная версия
void call_as_ver3()
{
    struct ver3 s;
    s.SizeOfStruct=sizeof(s);
    s.coord_X=123;
    s.coord_Y=456;
    s.radius=10;
    s.color=1;
    s.fill_brush_type=3;
    printf ("** %s()\n", __FUNCTION__);
    draw_circle(&s);
};

int main()
{
    call_as_ver1();
    call_as_ver2();
    call_as_ver3();
};

```

Другими словами, поле *SizeOfStruct* берет на себя роль поля *версия структуры*. Это может быть перечисляемый тип (1, 2, 3, и т. д.), но установка поля *SizeOfStruct* равным *sizeof(struct...)*, это лучше защищено от ошибок: в вызываемом коде мы просто пишем *s.SizeOfStruct=sizeof(...)*.

В Си++ эта проблема решается *наследованием* (?? (стр. ??)). Просто расширяете ваш базовый класс (назовем его *Circle*), и затем вам нужен *ColoredCircle*, а потом *FilledColoredCircle*, и так далее. Текущая *версия* объекта (или более точно, текущий *тип*) будет определяться при помощи **RTTI!** в Си++.

Так что если вы где-то в **MSDN!** видите *SizeOfStruct* — вероятно, эта структура уже расширялась в прошлом, как минимум один раз.

### 3.23.4. Файл с рекордами в игре «Block out» и примитивная сериализация

Многие видеоигры имеют файл с рекордами, иногда называемый «Зал славы». Древняя игра «Block out»<sup>50</sup> (трехмерный тетрис из 1989) не исключение, вот что мы можем увидеть в конце:

<sup>50</sup><http://www.bestoldgames.net/eng/old-games/blockout.php>



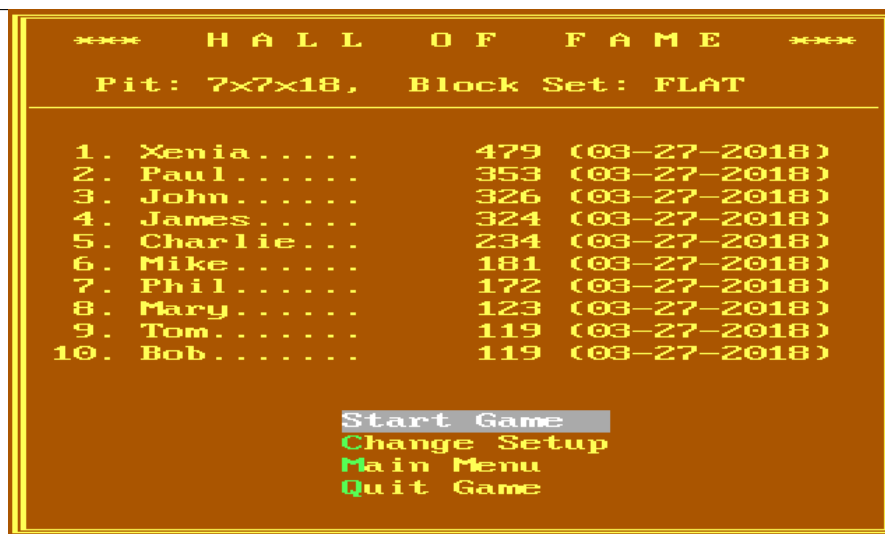


Рис. 3.4: Таблица рекордов

Мы можем увидеть, что после того как мы всякий раз добавляем свое имя, этот файл меняется: *BLSCORE.DAT*.

```
% xxd -g 1 BLSCORE.DAT
00000000: 0a 00 58 65 6e 69 61 2e 2e 2e 2e 00 df 01 00 ..Xenia.....
00000010: 00 30 33 2d 32 37 2d 32 30 31 38 00 50 61 75 6c .03-27-2018.Paul
00000020: 2e 2e 2e 2e 2e 00 61 01 00 00 30 33 2d 32 37 .....a...03-27
00000030: 2d 32 30 31 38 00 4a 6f 68 6e 2e 2e 2e 2e 2e 2e -2018.John.....
00000040: 00 46 01 00 00 30 33 2d 32 37 2d 32 30 31 38 00 .F...03-27-2018.
00000050: 4a 61 6d 65 73 2e 2e 2e 2e 00 44 01 00 00 30 James.....D...0
00000060: 33 2d 32 37 2d 32 30 31 38 00 43 68 61 72 6c 69 3-27-2018.Charli
00000070: 65 2e 2e 2e 00 ea 00 00 00 30 33 2d 32 37 2d 32 e.....03-27-2
00000080: 30 31 38 00 4d 69 6b 65 2e 2e 2e 2e 2e 2e 00 b5 018.Mike.....
00000090: 00 00 00 30 33 2d 32 37 2d 32 30 31 38 00 50 68 ...03-27-2018.Ph
000000a0: 69 6c 2e 2e 2e 2e 2e 00 ac 00 00 00 30 33 2d il.....03-
000000b0: 32 37 2d 32 30 31 38 00 4d 61 72 79 2e 2e 2e 2e 27-2018.Mary....
000000c0: 2e 2e 00 7b 00 00 00 30 33 2d 32 37 2d 32 30 31 ...{...03-27-201
000000d0: 38 00 54 6f 6d 2e 2e 2e 2e 2e 2e 00 77 00 00 8.Tom.....w..
000000e0: 00 30 33 2d 32 37 2d 32 30 31 38 00 42 6f 62 2e .03-27-2018.Bob.
000000f0: 2e 2e 2e 2e 2e 00 77 00 00 00 30 33 2d 32 37 .....w...03-27
00000100: 2d 32 30 31 38 00 -2018.
```

Все записи и так хорошо видны. Самый первый байт, вероятно, это количество записей. Второй это 0, и, на самом деле, число записей может быть 16-битным значением, которое простирается на 2 байта.

После имени «Xenia» мы видим байты 0xDF и 0x01. У Xenia 479 очков, и это именно 0x1DF в шестнадцатеричной системе. Так что значение рекорда, вероятно, 16-битное целочисленное, а может и 32-битное: после каждого по два нулевых байта.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Подумаем теперь о том факте, что и элементы массива, и элементы структуры всегда располагаются в памяти друг к другу вплотную. Это позволяет там записывать весь массив/структуру в файл используя простую ф-цию *write()* или *fwrite()*, а затем восстанавливать его используя *read()* или *fread()*, настолько всё просто. Это то, что сейчас называется *сериализацией*.

## Чтение

Напишем программу на Си для чтения файла рекордов:

```
#include <assert.h>
#include <stdio.h>
#include <stdint.h>
#include <string.h>

struct entry
{
    char name[11]; // включая терминирующий ноль
    uint32_t score;
    char date[11]; // включая терминирующий ноль
} __attribute__((aligned (1),packed));

struct highscore_file
{
    uint8_t count;
    uint8_t unknown;
    struct entry entries[10];
} __attribute__((aligned (1), packed));

struct highscore_file file;

int main(int argc, char* argv[])
{
    FILE* f=fopen(argv[1], "rb");
    assert (f!=NULL);
    size_t got=fread(&file, 1, sizeof(struct highscore_file), f);
    assert (got==sizeof(struct highscore_file));
    fclose(f);
    for (int i=0; i<file.count; i++)
    {
        printf ("name=%s score=%d date=%s\n",
                file.entries[i].name,
                file.entries[i].score,
                file.entries[i].date);
    };
};
```

Нужно добавить атрибут GCC (*aligned (1),packed*), чтобы все поля структуры были упакованы по 1-байтной границе.

Конечно, это работает:

```
name=Xenia..... score=479 date=03-27-2018
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
name=Paul..... score=353 date=03-27-2018
name=John..... score=326 date=03-27-2018
name=James..... score=324 date=03-27-2018
name=Charlie... score=234 date=03-27-2018
name=Mike..... score=181 date=03-27-2018
name=Phil..... score=172 date=03-27-2018
name=Mary..... score=123 date=03-27-2018
name=Tom..... score=119 date=03-27-2018
name=Bob..... score=119 date=03-27-2018
```

(Нужно добавить, что каждое имя дополнено точками, и на экране, и в файле, вероятно, с эстетическими целями.)

### Запись

Посмотрим, правы ли мы насчет длины значения очков. Действительно ли там 32 бита?

```
int main(int argc, char* argv[])
{
    FILE* f=fopen(argv[1], "rb");
    assert (f!=NULL);
    size_t got=fread(&file, 1, sizeof(struct highscore_file), f);
    assert (got==sizeof(struct highscore_file));
    fclose(f);

    strcpy (file.entries[1].name, "Mallory...");
    file.entries[1].score=12345678;
    strcpy (file.entries[1].date, "08-12-2016");

    f=fopen(argv[1], "wb");
    assert (f!=NULL);
    got=fwrite(&file, 1, sizeof(struct highscore_file), f);
    assert (got==sizeof(struct highscore_file));
    fclose(f);
};
```

Запустим Blockout:

```

***  H A L L  O F  F A M E  ***
Pit: 7x7x18,  Block Set: FLAT

1. Xenia.....      479 (03-27-2018)
2. Mallory... 345678 (08-12-2016)
3. John.....      326 (03-27-2018)
4. James.....      324 (03-27-2018)
5. Charlie...     234 (03-27-2018)
6. Mike.....      181 (03-27-2018)
7. Phil.....      172 (03-27-2018)
8. Mary.....      123 (03-27-2018)
9. _____      123 (03-27-2018)
10. Tom.....      119 (03-27-2018)

```

Рис. 3.5: Таблица рекордов

Первые две цифры (1 или 2) пропали: 12345678 стало 345678. Вероятно, это проблемы с форматированием... но число почти корректно. Заменяю на 999999 и запускаю снова:

```

***  H A L L  O F  F A M E  ***
Pit: 7x7x18,  Block Set: FLAT

1. Xenia.....      479 (03-27-2018)
2. Mallory... 999999 (08-12-2016)
3. John.....      326 (03-27-2018)
4. James.....      324 (03-27-2018)
5. Charlie...     234 (03-27-2018)
6. Mike.....      181 (03-27-2018)
7. Phil.....      172 (03-27-2018)
8. _____      133 (03-27-2018)
9. _____      132 (03-27-2018)
10. Mary.....      123 (03-27-2018)

```

Рис. 3.6: Таблица рекордов

Теперь всё верно. Да, значение очков это 32-битное целочисленное.

### Это сериализация?

...почти. Сериализация как эта очень популярна в научном и инженерном ПО, где скорость намного важнее чем конвертирование в **XML!**<sup>51</sup> или **JSON!**<sup>52</sup> и

<sup>51</sup>**XML!**

<sup>52</sup>**JSON!**

назад.

Одна очевидная вещь это то что вы, разумеется, не можете сериализовать указатели, потому что каждый раз, когда вы загружаете файл в память, все структуры могут быть размещены в других местах.

Но: если вы работаете на каком-нибудь маломощном **MCU!** с простой **OS!** на нем, и все ваши структуры всегда расположены в тех же местах в памяти, тогда, вы можете сохранять и восстанавливать указатели.

### Случайный шум

Когда я готовил этот пример, я запускал «Block out» много раз и немного играл, чтобы заполнить таблицу рекордов случайными именами.

И когда было только 3 записи в файле, я увидел это:

```

00000000: 03 00 54 6f 6d 61 73 2e 2e 2e 2e 00 da 2a 00 ..Tomas.....*.
00000010: 00 30 38 2d 31 32 2d 32 30 31 36 00 43 68 61 72 .08-12-2016.Char
00000020: 6c 69 65 2e 2e 2e 00 8b 1e 00 00 30 38 2d 31 32 lie.....08-12
00000030: 2d 32 30 31 36 00 4a 6f 68 6e 2e 2e 2e 2e 2e 2e -2016.John.....
00000040: 00 80 00 00 00 30 38 2d 31 32 2d 32 30 31 36 00 .....08-12-2016.
00000050: 00 00 57 c8 a2 01 06 01 ba f9 47 c7 05 00 f8 4f ..W.....G....0
00000060: 06 01 06 01 a6 32 00 00 00 00 00 00 00 00 00 00 .....2.....
00000070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000a0: 00 00 00 00 00 00 00 00 00 00 93 c6 a2 01 46 72 .....Fr
000000b0: 8c f9 f6 c5 05 00 f8 4f 00 02 06 01 a6 32 06 01 .....0....2..
000000c0: 00 00 98 f9 f2 c0 05 00 f8 4f 00 02 a6 32 a2 f9 .....0...2..
000000d0: 80 c1 a6 32 a6 32 f4 4f aa f9 39 c1 a6 32 06 01 ...2.2.0..9..2..
000000e0: b4 f9 2b c5 a6 32 e1 4f c7 c8 a2 01 82 72 c6 f9 ..+..2.0.....r..
000000f0: 30 c0 05 00 00 00 00 00 00 00 a6 32 d4 f9 76 2d 0.....2..v-
00000100: a6 32 00 00 00 00 .....2....

```

Первый байт это 3, означая, что здесь 3 записи. И присутствуют 3 записи. Но затем мы видим случайный шум во второй части файла.

Шум, вероятно, связан с неинициализированными данными. Вероятно, «Block out» выделил память для 10 записей где-то в [куче](#), где, очевидно, присутствует некоторый псевдослучайный шум (оставшийся от чего-то еще). Затем он выставил первый/второй байт, заполнил 3 записи и затем он никогда не трогал оставшиеся 7 записей, так что они были записаны в файл как есть.

Когда «Block out», при следующем запуске, загружает файл с рекордами, он читает кол-во записей из первого/второго байта (3) и полностью игнорирует всё, что идет после.

Это распространенная проблема. Вернее, не совсем проблема в строгом смысле: ничего не глючит, но лишняя информация может попадать наружу.

Microsoft Word версий 90-х часто оставлял куски ранее отредактированных текстов в файлах \*.doc\*. В те времена это было что-то вроде развлечения, полу-

читать *.doc*-файл от кого-то, открыть его в шестнадцатеричном редакторе и прочитать что-то еще, что редактировалось на том компьютере до этого.

Эта проблема может быть куда более серьезная: ошибка Heartbleed в OpenSSL.

### Домашнее задание

«Block out» поддерживает несколько поликубов (flat/basic/extended), размер стакана можно конфигурировать, и т. д. И похоже на то, что для каждой конфигурации, «Block out» имеет свою таблицу рекордов. Я заметил, что некоторая информация вероятно сохраняется в файле *BLSCORE.IDX*. Это может быть домашнее задание для хардкорных фанатов «Block out» — разобраться также и в этой структуре.

Файлы «Block out» здесь: <http://beginners.re/examples/blockout.zip> (включая двоичные файлы с рекордами, которые я использовал в этом примере). Для запуска можно использовать DosBox.

## 3.24. memmove() и memcpy()

Разница между этими стандартными ф-циями в том, что *memcpy()* слепо копирует блок в другое место, в то время как *memmove()* корректно обрабатывает блоки, хранимые внахлест. Например, вы хотите оттащить строку на два байта назад:

```
`|.|.|h|e|l|l|o|...` -> `|h|e|l|l|o|...`
```

*memcpy()*, которая копирует 32-битные или 64-битные слова за раз, или даже **SIMD!**, здесь очевидно не сработают, потому как нужно использовать ф-цию копирования работающую побайтово.

Теперь даже более сложный пример, вставьте 2 байта впереди строки:

```
`|h|e|l|l|o|...` -> `|.|.|h|e|l|l|o|...`
```

Теперь даже ф-ция работающая побайтово не сработает, нужно копировать байты с конца.

Это тот редкий случай, когда x86 флаг DF нужно выставлять перед инструкцией REP MOVSB: DF определяет направление, и теперь мы должны двигаться назад.

Обычная процедура *memmove()* работает примерно так: 1) если источник ниже назначения, копируем вперед; 2) если источник над назначением, копируем назад.

Это *memmove()* из uClibc:

```
void *memmove(void *dest, const void *src, size_t n)
{
    int eax, ecx, esi, edi;
    __asm__ __volatile__(
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

"    movl    %%eax, %%edi\n"
"    cmpl   %%esi, %%eax\n"
"    je     2f\n" /* (optional) src == dest -> NOP */
"    jb     1f\n" /* src > dest -> simple copy */
"    leal   -1(%%esi,%%ecx), %%esi\n"
"    leal   -1(%%eax,%%ecx), %%edi\n"
"    std\n"
"1:    rep; movsb\n"
"    cld\n"
"2:\n"
: "=&c" (ecx), "=&S" (esi), "=&a" (eax), "=&D" (edi)
: "0" (n), "1" (src), "2" (dest)
: "memory"
);
return (void*)eax;
}

```

В первом случае, REP MOVSB вызывается со сброшенным флагом DF. Во втором, DF в начале выставляется, но потом сбрасывается.

Более сложный алгоритм имеет такую часть:

«если разница между *источником* и *назначением* больше чем ширина слова, копируем используя слова нежели байты, и используем побитовое копирование для копирования невыровненных частей».

Так происходит в неоптимизированной части на Си в Glibc 2.24.

Учитывая всё это, *memmove()* может работать медленнее, чем *memcpy()*. Но некоторые люди, включая Линуса Торвальдса, спорят<sup>53</sup> что *memcpy()* должна быть синонимом *memmove()*, а последняя ф-ция должна в начале проверять, пересекаются ли буферы или нет, и затем вести себя как *memcpy()* или *memmove()*. Все же, в наше время, проверка на пересекающиеся буферы это очень дешевая операция.

### 3.24.1. Анти-отладочный прием

Я слышал об анти-отладочном приеме, где всё что вам нужно для падения процесса это выставить DF: следующий вызов *memcpy()* приведет к падению, потому что будет копировать назад. Но я не могу это проверить: похоже, все процедуры копирования сбрасывают/выставляют DF, как им надо. С другой стороны, *memmove()* из uClibc, код которой я цитировал здесь, не имеет явного сброса DF (он подразумевает, что DF всегда сброшен?), так что он может и упасть.

## 3.25. setjmp/longjmp

setjmp/longjmp это механизм в Си, очень похожий на throw/catch в Си++ и других высокоуровневых **PL!**. Вот пример из zlib:

<sup>53</sup>[https://bugzilla.redhat.com/show\\_bug.cgi?id=638477#c132](https://bugzilla.redhat.com/show_bug.cgi?id=638477#c132)

```

...

/* return if bits() or decode() tries to read past available input */
if (setjmp(s.env) != 0) /* if came back here via longjmp(), ↵
↳ */
    err = 2; /* then skip decomp(), return ↵
↳ error */
else
    err = decomp(&s); /* decompress */

...

/* load at least need bits into val */
val = s->bitbuf;
while (s->bitcnt < need) {
    if (s->left == 0) {
        s->left = s->infun(s->inhow, &(s->in));
        if (s->left == 0) longjmp(s->env, 1); /* out of input */
    }

...

    if (s->left == 0) {
        s->left = s->infun(s->inhow, &(s->in));
        if (s->left == 0) longjmp(s->env, 1); /* out of input */
    }
}

```

( zlib/contrib/blast/blast.c )

Вызов `setjmp()` сохраняет текущие **PC!**, **SP!** и другие регистры в структуре `env`, затем возвращает 0.

В случае ошибки, `longjmp()` телепортирует вас в место точно после вызова `setjmp()`, как если бы вызов `setjmp()` вернул ненулевое значение (которое было передано в `longjmp()`). Это напоминает нам сисколл `fork()` в UNIX.

Посмотрим в более дистиллированный пример:

```

#include <stdio.h>
#include <setjmp.h>

jmp_buf env;

void f2()
{
    printf ("%s() begin\n", __FUNCTION__);
    // something odd happened here
    longjmp (env, 1234);
    printf ("%s() end\n", __FUNCTION__);
};

void f1()
{
    printf ("%s() begin\n", __FUNCTION__);
    f2();
    printf ("%s() end\n", __FUNCTION__);
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

};

int main()
{
    int err=setjmp(env);
    if (err==0)
    {
        f1();
    }
    else
    {
        printf ("Error %d\n", err);
    }
};

```

Если запустим, то увидим:

```

f1() begin
f2() begin
Error 1234

```

Структура `jmp_buf` обычно недокументирована, чтобы сохранить прямую совместимость.

Посмотрим, как `setjmp()` реализован в MSVC 2013 x64:

```

...
; RCX = address of jmp_buf

mov     [rcx], rax
mov     [rcx+8], rbx
mov     [rcx+18h], rbp
mov     [rcx+20h], rsi
mov     [rcx+28h], rdi
mov     [rcx+30h], r12
mov     [rcx+38h], r13
mov     [rcx+40h], r14
mov     [rcx+48h], r15
lea     r8, [rsp+arg_0]
mov     [rcx+10h], r8
mov     r8, [rsp+0]      ; get saved RA from stack
mov     [rcx+50h], r8   ; save it
stmxcsr dword ptr [rcx+58h]
fnstcw word ptr [rcx+5Ch]
movdqa xmmword ptr [rcx+60h], xmm6
movdqa xmmword ptr [rcx+70h], xmm7
movdqa xmmword ptr [rcx+80h], xmm8
movdqa xmmword ptr [rcx+90h], xmm9
movdqa xmmword ptr [rcx+0A0h], xmm10
movdqa xmmword ptr [rcx+0B0h], xmm11
movdqa xmmword ptr [rcx+0C0h], xmm12
movdqa xmmword ptr [rcx+0D0h], xmm13

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

movdqa xmmword ptr [rcx+0E0h], xmm14
movdqa xmmword ptr [rcx+0F0h], xmm15
retn

```

Она просто заполняет структуру `jmp_buf` текущими значениями почти всех регистров. Также, текущее значение **RA!** берется из стека и сохраняется в `jmp_buf`: В будущем, оно будет использовано как новое значение **PC!**.

Теперь `longjmp()`:

```

...

; RCX = address of jmp_buf

mov     rax, rdx
mov     rbx, [rcx+8]
mov     rsi, [rcx+20h]
mov     rdi, [rcx+28h]
mov     r12, [rcx+30h]
mov     r13, [rcx+38h]
mov     r14, [rcx+40h]
mov     r15, [rcx+48h]
ldmxcsr dword ptr [rcx+58h]
fnclex
fldcw   word ptr [rcx+5Ch]
movdqa xmm6, xmmword ptr [rcx+60h]
movdqa xmm7, xmmword ptr [rcx+70h]
movdqa xmm8, xmmword ptr [rcx+80h]
movdqa xmm9, xmmword ptr [rcx+90h]
movdqa xmm10, xmmword ptr [rcx+0A0h]
movdqa xmm11, xmmword ptr [rcx+0B0h]
movdqa xmm12, xmmword ptr [rcx+0C0h]
movdqa xmm13, xmmword ptr [rcx+0D0h]
movdqa xmm14, xmmword ptr [rcx+0E0h]
movdqa xmm15, xmmword ptr [rcx+0F0h]
mov     rdx, [rcx+50h] ; get PC (RIP)
mov     rbp, [rcx+18h]
mov     rsp, [rcx+10h]
jmp     rdx           ; jump to saved PC

...

```

Она просто восстанавливает (почти) все регистры, берет из структуры **RA!** и переходит туда. Эффект такой же, как если бы `setjmp()` вернула управление в вызывающую ф-цию. Также, `RAX` выставляется такой же, как и второй аргумент `longjmp()`. Это работает, как если бы `setjmp()` вернуло ненулевое значение в самом начале.

Как побочный эффект восстановления **SPI**, все значения в стеке, которые были установлены и использованы между вызовами `setjmp()` и `longjmp()`, просто выкидываются. Они больше не будут использоваться. Следовательно, `longjmp()`

обычно делает переход назад<sup>54</sup>.

Это подразумевает, что в отличие от механизма throw/catch в Си++, память не будет освобождаться, деструкторы не будут вызываться, и т. д. Следовательно, эта техника иногда опасна. Тем не менее, всё это довольно популярно, до сих пор. Это все еще используется в Oracle RDBMS.

Это также имеет неожиданный побочный эффект: если некий буфер был перезаписан внутри ф-ции (может даже из-за удаленной атаки), и ф-ция хочет сообщить об ошибке, и вызывает longjmp(), перезаписанная часть стека становится просто неиспользованной.

В качестве упражнения, попробуйте понять, почему не все регистры сохраняются. Почему пропускаются регистры XMM0-XMM5 и другие?

## 3.26. Другие нездоровые хаки связанные со стеком

### 3.26.1. Доступ к аргументам и локальным переменным вызывающей ф-ции

Из основ Си/Си++мы знаем, что иметь доступ к аргументам ф-ции или её локальным переменным — невозможно.

Тем не менее, при помощи грязных хаков это возможно. Например:

```
#include <stdio.h>

void f(char *text)
{
    // распечатать стек
    int *tmp=&text;
    for (int i=0; i<20; i++)
    {
        printf ("0x%x\n", *tmp);
        tmp++;
    };
};

void draw_text(int X, int Y, char* text)
{
    f(text);

    printf ("Собираемся нарисовать [%s] на %d:%d\n", text, X, Y);
};

int main()
{
    printf ("адрес main()=0x%x\n", &main);
};
```

<sup>54</sup>Впрочем, существуют люди, которые используют всё это для куда более сложных вещей, включая имитацию копроцедур, и т. д.: <https://www.embeddedrelated.com/showarticle/455.php>, <http://fanf.livejournal.com/105413.html>

```

printf ("адрес draw_text()=0x%x\n", &draw_text);
draw_text(100, 200, "Hello!");
};

```

На 32-битной Ubuntu 16.04 и GCC 5.4.0, я получил это:

```

адрес main()=0x80484f8
адрес draw_text()=0x80484cb
0x8048645      первый аргумент f()
0x8048628
0xbf8ab98
0xb7634590
0xb779eddc
0xb77e4918
0xbf8aba8
0x8048547      адрес возврата в середину main()
0x64          первый аргумент draw_text()
0xc8          второй аргумент draw_text()
0x8048645      третий аргумент draw_text()
0x8048581
0xb779d3dc
0xbf8abc0
0x0
0xb7603637
0xb779d000
0xb779d000
0x0
0xb7603637

```

(Комментарии мои.)

Так как *f()* начинает перебирать элементы стека начиная со своего первого аргумента, первый элемент стека это действительно указатель на строку «Hello!». Мы видим что её адрес также используется как третий аргумент для ф-ции *draw\_text()*.

В *f()* мы можем читать аргументы и локальные переменные ф-ций, если мы точно знаем разметку стека, но она все время меняется, от компилятора к компилятору. Различные уровни оптимизаций также сильно влияют на разметку.

Но если мы можем каким-то образом распознать нужную нам информацию, мы даже можем модифицировать её. Как пример, я переработаю ф-цию *f()*:

```

void f(char *text)
{
    ...

    // найти пару переменных 100, 200 и модифицировать вторую
    tmp=&text;
    for (int i=0; i<20; i++)
    {
        if (*tmp==100 && *(tmp+1)==200)
        {
            printf ("нашли\n");
        }
    }
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        *(tmp+1)=210; // поменять 200 на 210
        break;
    };
    tmp++;
};
};

```

Таки работает:

```

нашли
Собираемся нарисовать [Hello!] на 100:210

```

## Итог

Это экстремально грязный хак, предназначенный для демонстрации внутренних стека. я никогда даже не видел и не слышал чтобы кто-то использовал такое в реальном коде. Но это, как всегда, хороший пример.

## Упражнение

Этот пример был скомпилирован без оптимизации на 32-битной Ubuntu используя GCC 5.4.0 и он работает. Но когда я включил максимальную оптимизацию (-O3), всё перестало работать. Попробуйте разобраться, почему.

Используйте свой любимый компилятор и OS, попробуйте разные уровни оптимизации, узнайте, заработает или нет, если нет, попробуйте понять, почему.

## 3.26.2. Возврат строки

Классическая ошибка из Brian W. Kernighan, Rob Pike, *Practice of Programming*, (1999):

```

#include <stdio.h>

char* amsg(int n, char* s)
{
    char buf[100];

    sprintf (buf, "error %d: %s\n", n, s) ;

    return buf;
};

int main()
{
    printf ("%s\n", amsg (1234, "something wrong!"));
};

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Она упадет. В начале, попытаемся понять, почему.

Это состояние стека перед возвратом из `amsg()`:

```
(низкие адреса)

...

[amsg(): 100 байт]
[RA] <- текущий SP
[два аргумента amsg]
[что-то еще]
[локальные переменные main()]

...

(высокие адреса)
```

Когда управление возвращается из `amsg()` в `main()`, пока всё хорошо. Но когда `printf()` вызывается из `main()`, который, в свою очередь, использует стек для своих нужд, затирая 100-байтный буфер. В лучшем случае, будет выведен случайный мусор.

Трудно поверить, но я знаю, как это исправить:

```
#include <stdio.h>

char* amsg(int n, char* s)
{
    char buf[100];

    sprintf (buf, "error %d: %s\n", n, s) ;

    return buf;
};

char* interim (int n, char* s)
{
    char large_buf[8000];
    // используем локальный массив.
    // а иначе компилятор выбросит его при оптимизации, как
    // неиспользуемый.
    large_buf[0]=0;
    return amsg (n, s);
};

int main()
{
    printf ("%s\n", interim (1234, "something wrong!"));
};
```

Это заработает если скомпилировано в MSVC 2013 без оптимизаций и с опцией `/GS-55`. MSVC предупредит: “warning C4172: returning address of local variable or

<sup>55</sup>Выключить защиту от переполнения буфера

temporary”, но код запустится и сообщение выведется. Посмотрим состояние стека в момент, когда `amsg()` возвращает управление в `interim()`:

```
(низкие адреса)
...
[amsg(): 100 байт]
[RA] <- текущий SP
[два аргумента amsg()]
[владения interim(), включая 8000 байт]
[еще что-то]
[локальные переменные main()]
...
(высокие адреса)
```

Теперь состояние стека на момент, когда `interim()` возвращает управление в `main()`:

```
(низкие адреса)
...
[amsg(): 100 байт]
[RA]
[два аргумента amsg()]
[владения interim(), включая 8000 байт]
[еще что-то] <- текущий SP
[локальные переменные main()]
...
(высокие адреса)
```

Так что когда `main()` вызывает `printf()`, он использует стек в месте, где выделен буфер в `interim()`, и не затирает 100 байт с сообщением об ошибке внутри, потому что 8000 байт (или может быть меньше) это достаточно для всего, что делает `printf()` и другие нисходящие ф-ции!

Это также может сработать, если между ними много ф-ций, например: `main() → f1() → f2() → f3() ... → amsg()`, и тогда результат `amsg()` используется в `main()`. Дистанция между **SP!** в `main()` и адресом буфера `buf[]` должна быть достаточно длинной.

Вот почему такие ошибки опасны: иногда ваш код работает (и бага прячется незамеченной), иногда нет. Такие баги в шутку называют *хейзенбаги* или *шрёдинбаги*.

## 3.27. OpenMP

OpenMP это один из простейших способов распараллелировать работу простого алгоритма.

В качестве примера, попытаемся написать программу для вычисления криптографического *nonce*. В моем простейшем примере, *nonce* это число, добавляемое к нешифрованному тексту, чтобы получить хэш с какой-то особенностью. Например, на одной из стадии, протокол Bitcoin требует найти такую *nonce*, чтобы в результате хэширования подряд шли определенное количество нулей.

Это еще называется *proof of work* (т.е. система доказывает, что она произвела какие-то очень ресурсоёмкие вычисления и затратила время на это).

Мой пример не связан с Bitcoin, он будет пытаться добавлять числа к строке «hello, world!» чтобы найти такое число, при котором строка вида «hello, world!\_<number>» после хэширования алгоритмом SHA512 будет содержать как минимум 3 нулевых байта в начале.

Ограничимся перебором всех чисел в интервале 0..INT32\_MAX-1 (т.е., 0x7FFFFFFE или 2147483646).

Алгоритм очень простой:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include "sha512.h"

int found=0;
int32_t checked=0;

int32_t* __min;
int32_t* __max;

time_t start;

#ifdef __GNUC__
#define min(X,Y) ((X) < (Y) ? (X) : (Y))
#define max(X,Y) ((X) > (Y) ? (X) : (Y))
#endif

void check_nonce (int32_t nonce)
{
    uint8_t buf[32];
    struct sha512_ctx ctx;
    uint8_t res[64];

    // update statistics
    int t=omp_get_thread_num();

    if (__min[t]==-1)
        __min[t]=nonce;
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

    if (__max[t]==-1)
        __max[t]=nonce;

    __min[t]=min(__min[t], nonce);
    __max[t]=max(__max[t], nonce);

    // idle if valid nonce found
    if (found)
        return;

    memset (buf, 0, sizeof(buf));
    sprintf (buf, "hello, world!_%d", nonce);

    sha512_init_ctx (&ctx);
    sha512_process_bytes (buf, strlen(buf), &ctx);
    sha512_finish_ctx (&ctx, &res);
    if (res[0]==0 && res[1]==0 && res[2]==0)
    {
        printf ("found (thread %d): [%s]. seconds spent=%d\n", t, ↵
        ↵ buf, time(NULL)-start);
        found=1;
    };
    #pragma omp atomic
    checked++;

    #pragma omp critical
    if ((checked % 100000)==0)
        printf ("checked=%d\n", checked);
};

int main()
{
    int32_t i;
    int threads=omp_get_max_threads();
    printf ("threads=%d\n", threads);

    __min=(int32_t*)malloc(threads*sizeof(int32_t));
    __max=(int32_t*)malloc(threads*sizeof(int32_t));
    for (i=0; i<threads; i++)
        __min[i]=__max[i]=-1;

    start=time(NULL);

    #pragma omp parallel for
    for (i=0; i<INT32_MAX; i++)
        check_nonce (i);

    for (i=0; i<threads; i++)
        printf ("__min[%d]=0x%08x __max[%d]=0x%08x\n", i, __min[i], ↵
        ↵ i, __max[i]);

    free(__min); free(__max);
};

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

`check_nonce()` просто добавляет число к строке, хеширует алгоритмом SHA512 и проверяет 3 нулевых байта в начале.

Очень важная часть кода — это:

```
#pragma omp parallel for
for (i=0; i<INT32_MAX; i++)
    check_nonce (i);
```

Да, вот настолько просто, без `#pragma` мы просто вызываем `check_nonce()` для каждого числа от 0 до `INT32_MAX` (0x7fffffff или 2147483647). С `#pragma`, компилятор добавляет специальный код, который разрежет интервал цикла на меньшие интервалы, чтобы запустить их на доступных ядрах **CPU!** <sup>56</sup>.

Пример может быть скомпилирован <sup>57</sup> в MSVC 2012:

```
cl openmp_example.c sha512.obj /openmp /O1 /Zi /Faopenmp_example.asm
```

Или в GCC:

```
gcc -fopenmp 2.c sha512.c -S -masm=intel
```

### 3.27.1. MSVC

Вот как MSVC 2012 генерирует главный цикл:

Листинг 3.123: MSVC 2012

```
push    OFFSET _main$omp$1
push    0
push    1
call    __vcomp_fork
add     esp, 16
```

Функции с префиксом `vcomp` связаны с OpenMP и находятся в файле `vcomp*.dll`. Так что тут запускается группа тредов.

Посмотрим на `_main$omp$1`:

Листинг 3.124: MSVC 2012

```
$T1 = -8      ; size = 4
$T2 = -4      ; size = 4
_main$omp$1 PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    push    ecx
    push    esi
```

<sup>56</sup>N.B.: Это намеренно упрощенный пример, но на практике, применение OpenMP может быть труднее и сложнее

<sup>57</sup>файлы `sha512.c(h)` и `u64.h` можно взять из библиотеки OpenSSL: <http://www.openssl.org/source/>

```

    lea    eax, DWORD PTR $T2[ebp]
    push  eax
    lea    eax, DWORD PTR $T1[ebp]
    push  eax
    push  1
    push  1
    push  2147483646      ; 7fffffffH
    push  0
    call  __vcomp_for_static_simple_init
    mov   esi, DWORD PTR $T1[ebp]
    add   esp, 24
    jmp   SHORT $LN6@main$omp$1
$LL2@main$omp$1:
    push  esi
    call  _check_nonce
    pop   ecx
    inc   esi
$LN6@main$omp$1:
    cmp   esi, DWORD PTR $T2[ebp]
    jle   SHORT $LL2@main$omp$1
    call  __vcomp_for_static_end
    pop   esi
    leave
    ret   0
_main$omp$1 ENDP

```

Эта функция будет запущена  $n$  раз параллельно, где  $n$  это число ядер **CPU!** `vcomp_for_static_simple_init()` вычисляет интервал для конструкта `for()` для текущего треда, в зависимости от текущего номера треда. Значения начала и конца цикла записаны в локальных переменных `$T1` и `$T2`. Вы также можете заметить `7fffffffh` (или `2147483646`) как аргумент для функции `vcomp_for_static_simple_init()` — это количество итераций всего цикла, оно будет поделено на равные части.

Потом мы видим новый цикл с вызовом функции `check_nonce()` делающей всю работу.

Добавил также немного кода в начале функции `check_nonce()` для сбора статистики, с какими аргументами эта функция вызывалась.

Вот что мы видим если запустим:

```

threads=4
...
checked=2800000
checked=3000000
checked=3200000
checked=3300000
found (thread 3): [hello, world!_1611446522]. seconds spent=3
__min[0]=0x00000000 __max[0]=0x1fffffff
__min[1]=0x20000000 __max[1]=0x3fffffff
__min[2]=0x40000000 __max[2]=0x5fffffff
__min[3]=0x60000000 __max[3]=0x7fffffff

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Да, результат правильный, первые 3 байта это нули:

```
C:\...\sha512sum test
000000f4a8fac5a4ed38794da4c1e39f54279ad5d9bb3c5465cdf57adaf60403
df6e3fe6019f5764fc9975e505a7395fed780fee50eb38dd4c0279cb114672e2 *test
```

Оно требует  $\approx 2..3$  секунды на 4-х ядерном Intel Xeon E3-1220 3.10 GHz.

В task manager мы видим 5 тредов: один главный тред + 4 запущенных. Никаких оптимизаций не было сделано, чтобы оставить этот пример в как можно более простом виде. Но, наверное, этот алгоритм может работать быстрее. У моего **CPU!** 4 ядра, вот почему OpenMP запустил именно 4 тредов.

Глядя на таблицу статистики, можно легко увидеть, что цикл был разделен очень точно на 4 равных части. Ну хорошо, почти равных, если не учитывать последний бит.

Имеются также прагмы и для [атомарных операций](#).

Посмотрим, как вот этот код будет скомпилирован:

```
#pragma omp atomic
checked++;

#pragma omp critical
if ((checked % 100000)==0)
    printf ("checked=%d\n", checked);
```

Листинг 3.125: MSVC 2012

```
    push    edi
    push    OFFSET _checked
    call    __vcomp_atomic_add_i4
; Line 55
    push    OFFSET _$vcomp$critsect$
    call    __vcomp_enter_critsect
    add    esp, 12
; Line 56
    mov    ecx, DWORD PTR _checked
    mov    eax, ecx
    cdq
    mov    esi, 100000      ; 000186a0H
    idiv   esi
    test   edx, edx
    jne    SHORT $LN1@check_nonc
; Line 57
    push    ecx
    push    OFFSET ??_C@_0M@NPNHLI00@checked?$DN?$CFd?6?$AA@
    call    _printf
    pop    ecx
    pop    ecx
$LN1@check_nonc:
    push    DWORD PTR _$vcomp$critsect$
    call    __vcomp_leave_critsect
    pop    ecx
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Как выясняется, функция `vcomp_atomic_add_i4()` в `vcomp*.dll` это просто крохотная функция имеющая инструкцию `LOCK XADD`<sup>58</sup>.

`vcomp_enter_critsect()` в конце концов вызывает функцию `win32 API! EnterCriticalSection()`<sup>59</sup>.

### 3.27.2. GCC

GCC 4.8.1 выдает программу показывающую точно такую же таблицу со статистикой, так что, реализация GCC делит цикл на части точно так же.

Листинг 3.126: GCC 4.8.1

```
mov    edi, OFFSET FLAT:main._omp_fn.0
call   GOMP_parallel_start
mov    edi, 0
call   main._omp_fn.0
call   GOMP_parallel_end
```

В отличие от реализации MSVC, то, что делает код GCC, это запускает 3 треда, но также запускает четвертый прямо в текущем треде. Так что здесь всего 4 треда а не 5 как в случае с MSVC.

Вот функция `main._omp_fn.0`:

Листинг 3.127: GCC 4.8.1

```
main._omp_fn.0:
    push    rbp
    mov     rbp, rsp
    push    rbx
    sub     rsp, 40
    mov     QWORD PTR [rbp-40], rdi
    call    omp_get_num_threads
    mov     ebx, eax
    call    omp_get_thread_num
    mov     esi, eax
    mov     eax, 2147483647 ; 0x7FFFFFFF
    cdq
    idiv   ebx
    mov     ecx, eax
    mov     eax, 2147483647 ; 0x7FFFFFFF
    cdq
    idiv   ebx
    mov     eax, edx
    cmp     esi, eax
    jl     .L15
.L18:
    imul   esi, ecx
    mov     edx, esi
    add     eax, edx
    lea    ebx, [rax+rcx]
```

<sup>58</sup>О префиксе `LOCK` читайте больше: ?? (стр. ??)

<sup>59</sup>О критических секциях читайте больше тут: ?? (стр. ??)

```

        cmp     eax, ebx
        jge    .L14
        mov    DWORD PTR [rbp-20], eax
.L17:
        mov    eax, DWORD PTR [rbp-20]
        mov    edi, eax
        call  check_nonce
        add   DWORD PTR [rbp-20], 1
        cmp   DWORD PTR [rbp-20], ebx
        jl    .L17
        jmp   .L14
.L15:
        mov    eax, 0
        add   ecx, 1
        jmp   .L18
.L14:
        add   rsp, 40
        pop   rbx
        pop   rbp
        ret

```

Здесь мы видим это деление явно: вызывая `omp_get_num_threads()` и `omp_get_thread_num()` мы получаем количество запущенных тредов, а также номер текущего тредра, и затем определяем интервал цикла. И затем запускаем `check_nonce()`.

GCC также вставляет инструкцию `LOCK ADD` прямо в том месте кода, где MSVC сгенерировал вызов отдельной функции в DLL:

Листинг 3.128: GCC 4.8.1

```

lock add     DWORD PTR checked[rip], 1
call        GOMP_critical_start
mov         ecx, DWORD PTR checked[rip]
mov         edx, 351843721
mov         eax, ecx
imul        edx
sar         edx, 13
mov         eax, ecx
sar         eax, 31
sub         edx, eax
mov         eax, edx
imul        eax, eax, 100000
sub         ecx, eax
mov         eax, ecx
test        eax, eax
jne         .L7
mov         eax, DWORD PTR checked[rip]
mov         esi, eax
mov         edi, OFFSET FLAT:.LC2 ; "checked=%d\n"
mov         eax, 0
call        printf
.L7:
call        GOMP_critical_end

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Функции с префиксом GOMP это часть библиотеки GNU OpenMP. В отличие от `vcomp*.dll`, её исходный код свободно доступен: [GitHub](#).

## 3.28. Еще одна heisenbug-а

Иногда, переполнение массива (или буфера) может привести к *ошибке заборного столба (fencepost error)*:

```
#include <stdio.h>

int array1[128];
int important_var1;
int important_var2;
int important_var3;
int important_var4;
int important_var5;

int main()
{
    important_var1=1;
    important_var2=2;
    important_var3=3;
    important_var4=4;
    important_var5=5;

    array1[0]=123;
    array1[128]=456; // BUG

    printf ("important_var1=%d\n", important_var1);
    printf ("important_var2=%d\n", important_var2);
    printf ("important_var3=%d\n", important_var3);
    printf ("important_var4=%d\n", important_var4);
    printf ("important_var5=%d\n", important_var5);
};
```

Вот что выводится в моем случае (неоптимизирующий GCC 5.4 x86 на Linux):

```
important_var1=1
important_var2=456
important_var3=3
important_var4=4
important_var5=5
```

Как бывает, `important_var2` может быть расположена компилятором сразу за `array1[]`:

Листинг 3.129: `objdump -x`

0804a040 g	0 .bss	00000200	array1
...			
0804a240 g	0 .bss	00000004	important_var2
0804a244 g	0 .bss	00000004	important_var4

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

...
0804a248 g    0 .bss  00000004      important_var1
0804a24c g    0 .bss  00000004      important_var3
0804a250 g    0 .bss  00000004      important_var5

```

Другой компилятор может расположить переменные в другом порядке, и другая переменная затрется. Это также *heisenbug*-а (?? (стр. ??)) — ошибка может появиться или может оставаться незамеченной в зависимости от версии компилятора и флагов оптимизации.

Если все переменные и массивы расположены в локальном стеке, защита стека может сработать, а может и нет. Хотя, Valgrind может находить такие ошибки.

Еще один пример в этой книге (игра Angband): ?? (стр. ??).

### 3.29. Случай с забытым return

Еще раз вернемся к части “Попытка использовать результат функции возвращающей *void*” ?? (стр. ??).

Вот ошибка которую я однажды видел.

И это также еще и демонстрация того факта, что Си/Си++ возвращает значение в регистре EAX/RAX.

В коде вроде этого я забыл добавить return:

```

#include <stdio.h>
#include <stdlib.h>

struct color
{
    int R;
    int G;
    int B;
};

struct color* create_color (int R, int G, int B)
{
    struct color* rt=(struct color*)malloc(sizeof(struct color));

    rt->R=R;
    rt->G=G;
    rt->B=B;
    // здесь должен быть "return rt;"
};

int main()
{
    struct color* a=create_color(1,2,3);
    printf ("%d %d %d\n", a->R, a->G, a->B);
};

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



Неоптимизирующий GCC 5.4 молча компилирует это без всяких предупреждений. И код работает! Посмотрим, почему:

Листинг 3.130: Неоптимизирующий GCC 5.4

```
create_color:
    push    rbp
    mov     rbp, rsp
    sub    rsp, 32
    mov    DWORD PTR [rbp-20], edi
    mov    DWORD PTR [rbp-24], esi
    mov    DWORD PTR [rbp-28], edx
    mov    edi, 12
    call   malloc
; RAX = указатель на только что выделенный буфер
; теперь буфер заполняется R/G/B:
    mov    QWORD PTR [rbp-8], rax
    mov    rax, QWORD PTR [rbp-8]
    mov    edx, DWORD PTR [rbp-20]
    mov    DWORD PTR [rax], edx
    mov    rax, QWORD PTR [rbp-8]
    mov    edx, DWORD PTR [rbp-24]
    mov    DWORD PTR [rax+4], edx
    mov    rax, QWORD PTR [rbp-8]
    mov    edx, DWORD PTR [rbp-28]
    mov    DWORD PTR [rax+8], edx
    nop
    leave
; RAX не модифицировался вплоть до этого места!
    ret
```

Если я добавляю `return rt;`, только одна инструкция добавляется в конца, и она избыточна:

Листинг 3.131: Неоптимизирующий GCC 5.4

```
create_color:
    push    rbp
    mov     rbp, rsp
    sub    rsp, 32
    mov    DWORD PTR [rbp-20], edi
    mov    DWORD PTR [rbp-24], esi
    mov    DWORD PTR [rbp-28], edx
    mov    edi, 12
    call   malloc
; RAX = указатель на буфер
    mov    QWORD PTR [rbp-8], rax
    mov    rax, QWORD PTR [rbp-8]
    mov    edx, DWORD PTR [rbp-20]
    mov    DWORD PTR [rax], edx
    mov    rax, QWORD PTR [rbp-8]
    mov    edx, DWORD PTR [rbp-24]
    mov    DWORD PTR [rax+4], edx
    mov    rax, QWORD PTR [rbp-8]
    mov    edx, DWORD PTR [rbp-28]
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov     DWORD PTR [rax+8], edx
; перезагрузить указатель на буфер в RAX опять, и это избыточная операция
mov     rax, QWORD PTR [rbp-8] ; новая инструкция
leave
ret

```

Ошибки вроде этой очень опасны, иногда они появляются, иногда пропадают. Это как heisenbug-и.

Попробуем оптимизирующий GCC:

Листинг 3.132: Оптимизирующий GCC 5.4

```

create_color:
    rep ret

main:
    xor     eax, eax
; как если бы вызвалась ф-ция create_color() и вернула 0
    sub     rsp, 8
    mov     r8d, DWORD PTR ds:8
    mov     ecx, DWORD PTR [rax+4]
    mov     edx, DWORD PTR [rax]
    mov     esi, OFFSET FLAT:.LC1
    mov     edi, 1
    call   __printf_chk
    xor     eax, eax
    add     rsp, 8
    ret

```

Компилятор определяет, что ф-ция ничего не возвращает, и оптимизирует всю ф-цию. И он считает, что ф-ция возвращает 0 по умолчанию. Этот ноль затем используется как адрес структуры в main(). Конечно, код падает.

GCC в режиме C++ также не выдает никаких предупреждений.

Попробуем неоптимизирующий MSVC 2015 x86. Он предупреждает о проблеме:

```
c:\tmp\3.c(19) : warning C4716: 'create_color': must return a value
```

И генерирует код, который упадет:

Листинг 3.133: Неоптимизирующий MSVC 2015 x86

```

_rt$ = -4
_R$ = 8
_G$ = 12
_B$ = 16
_create_color PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    push    12
    call   _malloc

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; EAX = указатель на буфер
add     esp, 4
mov     DWORD PTR _rt$[ebp], eax
mov     eax, DWORD PTR _rt$[ebp]
mov     ecx, DWORD PTR _R$[ebp]
mov     DWORD PTR [eax], ecx
mov     edx, DWORD PTR _rt$[ebp]
mov     eax, DWORD PTR _G$[ebp]
; EAX указывает на аргумент G:
mov     DWORD PTR [edx+4], eax
mov     ecx, DWORD PTR _rt$[ebp]
mov     edx, DWORD PTR _B$[ebp]
mov     DWORD PTR [ecx+8], edx
mov     esp, ebp
pop     ebp
; EAX = G в этом месте:
ret     0
_create_color ENDP

```

Оптимизирующий MSVC 2015 x86 также генерирует падающий код, но по другой причине:

Листинг 3.134: Оптимизирующий MSVC 2015 x86

```

_a$ = -4
_main PROC
; здесь вставлен (inline) оптимизированный код create_color():
push   ecx
push   12
call   _malloc
mov    DWORD PTR [eax], 1
mov    DWORD PTR [eax+4], 2
mov    DWORD PTR [eax+8], 3
; EAX указывает на выделенный буфер, и он заполняется, OK
; теперь мы перезагружаем указатель на буфер, надеясь, что он в переменной
"а"
; но вставленная (inlined) ф-ция не сохраняла указатель в переменной "а"!
mov    eax, DWORD PTR _a$[esp+8]
; EAX = в этом месте, какой-нибудь случайный мусор
push   DWORD PTR [eax+8]
push   DWORD PTR [eax+4]
push   DWORD PTR [eax]
push   OFFSET $SG6074
call   _printf
xor    eax, eax
add    esp, 24
ret    0
_main ENDP

_R$ = 8
_G$ = 12
_B$ = 16
_create_color PROC
push   12

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

call    _malloc
mov     ecx, DWORD PTR _R$[esp]
add     esp, 4
mov     DWORD PTR [eax], ecx
mov     ecx, DWORD PTR _G$[esp-4]
mov     DWORD PTR [eax+4], ecx
mov     ecx, DWORD PTR _B$[esp-4]
mov     DWORD PTR [eax+8], ecx
; EAX указывает на выделенный буфер, OK
ret     0
_create_color ENDP

```

Хотя, неоптимизирующий MSVC 2015 x64 генерирует работающий код:

#### Листинг 3.135: Non-optimizing MSVC 2015 x64

```

rt$ = 32
R$ = 64
G$ = 72
B$ = 80
create_color PROC
    mov     DWORD PTR [rsp+24], r8d
    mov     DWORD PTR [rsp+16], edx
    mov     DWORD PTR [rsp+8], ecx
    sub     rsp, 56
    mov     ecx, 12
    call    malloc
; RAX = выделенный буфер
    mov     QWORD PTR rt$[rsp], rax
    mov     rax, QWORD PTR rt$[rsp]
    mov     ecx, DWORD PTR R$[rsp]
    mov     DWORD PTR [rax], ecx
    mov     rax, QWORD PTR rt$[rsp]
    mov     ecx, DWORD PTR G$[rsp]
    mov     DWORD PTR [rax+4], ecx
    mov     rax, QWORD PTR rt$[rsp]
    mov     ecx, DWORD PTR B$[rsp]
    mov     DWORD PTR [rax+8], ecx
    add     rsp, 56
; RAX не меняется вплоть до этого места
    ret     0
create_color ENDP

```

Оптимизирующий MSVC 2015 x64 также вставляет код ф-ции в другую ф-цию (inline), как в случае с x86, и итоговый код также падает.

Это реальный фрагмент из моей библиотеки *octothorpe*, который работал, и все тесты нормально проходили. Так оно и было, какое-то время без return..

```

uint32_t LPHM_u32_hash(void *key)
{
    jenkins_one_at_a_time_hash_u32((uint32_t)key);
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
}
```

Мораль истории: предупреждения очень важны, используйте `-Wall`, и т. д., и т. п. Без директивы `return`, компилятор может просто ничего не делать в этом месте.

Такая ошибка, если незамечена, может испортить целый день.

Также, *shotgun debugging* это плохо, опять же потому что такая ошибка может затеряться (“сейчас всё работает, ну пусть так оно и остается, ничего не надо менять”).

См. также: обсуждение на Hacker News<sup>60</sup> и архивированный пост в блоге<sup>61</sup>.

### 3.30. Домашнее задание: больше об указателях на ф-ции и объединениях (union)

Этот код был скопирован из *dwm*<sup>62</sup>, это, вероятно, самый маленький оконный менеджер под Linux.

Проблема: клавиши нажатые пользователем должны быть связаны с различными ф-циями внутри *dwm*. Обычно это решается в виде большого *switch()*.

Обыкновенно, это важная проблема, которую решают программисты любых видеоигр.

Вероятно, создатели *dwm* хотели сделать код ясным, и при этом, чтобы пользователи могли легко модифицировать его:

```
...

typedef union {
    int i;
    unsigned int ui;
    float f;
    const void *v;
} Arg;

...

typedef struct {
    unsigned int mod;
    KeySym keysym;
    void (*func)(const Arg *);
    const Arg arg;
} Key;
```

<sup>60</sup><https://news.ycombinator.com/item?id=18671609>

<sup>61</sup>[https://web.archive.org/web/20190317231721/https://yurichev.com/blog/no\\_return/](https://web.archive.org/web/20190317231721/https://yurichev.com/blog/no_return/)

<sup>62</sup><https://dwm.suckless.org/>

```

...
static Key keys[] = {
    /* modifier      key          function      argument */
    { MODKEY,        XK_p,        spawn,        {.v = dmenucmd } },
    { MODKEY|ShiftMask, XK_Return,  spawn,        {.v = termcmd } },
    { MODKEY,        XK_b,        togglebar,    {0} },
    { MODKEY,        XK_j,        focusstack,   {.i = +1 } },
    { MODKEY,        XK_k,        focusstack,   {.i = -1 } },
    { MODKEY,        XK_i,        incnmaster,   {.i = +1 } },
    { MODKEY,        XK_d,        incnmaster,   {.i = -1 } },
    { MODKEY,        XK_h,        setmfact,     {.f = -0.05} },
    { MODKEY,        XK_l,        setmfact,     {.f = +0.05} },
    { MODKEY,        XK_Return,  zoom,         {0} },
    { MODKEY,        XK_Tab,     view,         {0} },
    { MODKEY|ShiftMask, XK_c,        killclient,   {0} },
    { MODKEY,        XK_t,        setlayout,    {.v = &layouts[0]} },
    { MODKEY,        XK_f,        setlayout,    {.v = &layouts[1]} },
    { MODKEY,        XK_m,        setlayout,    {.v = &layouts[2]} },
    ...
}

void
spawn(const Arg *arg)
{
    ...
}

void
focusstack(const Arg *arg)
{
    ...
}

```

Для каждой клавиши с модификаторами (shift/ctrl/alt) определена ф-ция. И даже более того: параметры (или аргументы) передаются в ф-цию в каждом конкретном случае. Но параметры могут иметь разные типы. Так что здесь используется *union*. Значение необходимого типа записано в таблицу. Каждая ф-ция берет себе то, что ей нужно.

В качестве домашнего задания, попробуйте написать код вроде такого, или разберитесь, как в *dwm* передается *union* и как это обрабатывается соответствующими ф-циями.

### 3.31. Windows 16-bit

16-битные программы под Windows в наше время редки, хотя иногда можно поработать с ними, в смысле ретрокомпьютинга, либо которые защищены донглами (?? (стр. ??)).

16-битные версии Windows были вплоть до 3.11. 95/98/ME также поддерживает 16-битный код, как и все 32-битные OS линейки [Windows NT](#). 64-битные версии

[Windows NT](#) не поддерживают 16-битный код вообще.

Код напоминает тот что под MS-DOS.

Исполняемые файлы имеют NE-тип (так называемый «new executable»).

Все рассмотренные здесь примеры скомпилированы компилятором OpenWatcom 1.9 используя эти опции:

```
wcl.exe -i=C:/WATCOM/h/win/ -s -os -bt=windows -bcl=windows example.c
```

### 3.31.1. Пример#1

```
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow )
{
    MessageBeep(MB_ICONEXCLAMATION);
    return 0;
};
```

```
WinMain    proc near
           push    bp
           mov     bp, sp
           mov     ax, 30h ; '0' ; MB_ICONEXCLAMATION constant
           push   ax
           call   MESSAGEBEEP
           xor     ax, ax      ; return 0
           pop    bp
           retn   0Ah
WinMain    endp
```

Пока всё просто.

### 3.31.2. Пример #2

```
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow )
{
    MessageBox (NULL, "hello, world", "caption", MB_YESNOCANCEL);
    return 0;
};
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

WinMain      proc near
              push    bp
              mov     bp, sp
              xor     ax, ax           ; NULL
              push   ax
              push   ds
              mov     ax, offset aHelloWorld ; 0x18. "hello, world"
              push   ax
              push   ds
              mov     ax, offset aCaption ; 0x10. "caption"
              push   ax
              mov     ax, 3           ; MB_YESNOCANCEL
              push   ax
              call   MESSAGEBOX
              xor     ax, ax           ; return 0
              pop    bp
              retn   0Ah
WinMain      endp

dseg02:0010 aCaption      db 'caption',0
dseg02:0018 aHelloWorld  db 'hello, world',0

```

Пара важных моментов: соглашение о передаче аргументов здесь PASCAL: оно указывает что самый первый аргумент должен передаваться первым (MB\_YESNOCANCEL), а самый последний аргумент — последним (NULL). Это соглашение также указывает вызываемой функции восстановить [указатель стека](#): поэтому инструкция RETN имеет аргумент 0Ah означая что указатель нужно сдвинуть вперед на 10 байт во время возврата из функции . Это как stdcall (?? (стр. ??)), только аргументы передаются в «естественном» порядке.

Указатели передаются парами: сначала сегмент данных, потом указатель внутри сегмента . В этом примере только один сегмент, так что DS всегда указывает на сегмент данных в исполняемом файле.

### 3.31.3. Пример #3

```

#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow )
{
    int result=MessageBox (NULL, "hello, world", "caption", MB_YESNOCANCEL)↵
    ↵ ;

    if (result==IDCANCEL)
        MessageBox (NULL, "you pressed cancel", "caption", MB_OK);
    else if (result==IDYES)
        MessageBox (NULL, "you pressed yes", "caption", MB_OK);
    else if (result==IDNO)

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

        MessageBox (NULL, "you pressed no", "caption", MB_OK);
    return 0;
};

```

```

WinMain    proc near
            push    bp
            mov     bp, sp
            xor     ax, ax        ; NULL
            push   ax
            push   ds
            mov     ax, offset aHelloWorld ; "hello, world"
            push   ax
            push   ds
            mov     ax, offset aCaption ; "caption"
            push   ax
            mov     ax, 3        ; MB_YESNOCANCEL
            push   ax
            call   MESSAGEBOX
            cmp     ax, 2        ; IDCANCEL
            jnz    short loc_2F
            xor     ax, ax
            push   ax
            push   ds
            mov     ax, offset aYouPressedCanc ; "you pressed cancel"
            jmp     short loc_49
loc_2F:
            cmp     ax, 6        ; IDYES
            jnz    short loc_3D
            xor     ax, ax
            push   ax
            push   ds
            mov     ax, offset aYouPressedYes ; "you pressed yes"
            jmp     short loc_49
loc_3D:
            cmp     ax, 7        ; IDNO
            jnz    short loc_57
            xor     ax, ax
            push   ax
            push   ds
loc_49:
            mov     ax, offset aYouPressedNo ; "you pressed no"
            push   ax
            push   ds
            mov     ax, offset aCaption ; "caption"
            push   ax
            xor     ax, ax
            push   ax
            call   MESSAGEBOX
loc_57:
            xor     ax, ax
            pop    bp
            retn   0Ah

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
WinMain      endp
```

Немного расширенная версия примера из предыдущей секции .

### 3.31.4. Пример #4

```
#include <windows.h>

int PASCAL func1 (int a, int b, int c)
{
    return a*b+c;
};

long PASCAL func2 (long a, long b, long c)
{
    return a*b+c;
};

long PASCAL func3 (long a, long b, long c, int d)
{
    return a*b+c-d;
};

int PASCAL WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow )
{
    func1 (123, 456, 789);
    func2 (600000, 700000, 800000);
    func3 (600000, 700000, 800000, 123);
    return 0;
};
```

```
func1      proc near
c          = word ptr 4
b          = word ptr 6
a          = word ptr 8

            push    bp
            mov     bp, sp
            mov     ax, [bp+a]
            imul   [bp+b]
            add     ax, [bp+c]
            pop     bp
            retn   6
func1      endp

func2      proc near
arg_0      = word ptr 4
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

arg_2      = word ptr 6
arg_4      = word ptr 8
arg_6      = word ptr 0Ah
arg_8      = word ptr 0Ch
arg_A      = word ptr 0Eh

        push    bp
        mov     bp, sp
        mov     ax, [bp+arg_8]
        mov     dx, [bp+arg_A]
        mov     bx, [bp+arg_4]
        mov     cx, [bp+arg_6]
        call    sub_B2 ; long 32-bit multiplication
        add     ax, [bp+arg_0]
        adc     dx, [bp+arg_2]
        pop     bp
        retn    12
func2     endp

func3     proc near

arg_0      = word ptr 4
arg_2      = word ptr 6
arg_4      = word ptr 8
arg_6      = word ptr 0Ah
arg_8      = word ptr 0Ch
arg_A      = word ptr 0Eh
arg_C      = word ptr 10h

        push    bp
        mov     bp, sp
        mov     ax, [bp+arg_A]
        mov     dx, [bp+arg_C]
        mov     bx, [bp+arg_6]
        mov     cx, [bp+arg_8]
        call    sub_B2 ; long 32-bit multiplication
        mov     cx, [bp+arg_2]
        add     cx, ax
        mov     bx, [bp+arg_4]
        adc     bx, dx          ; BX=high part, CX=low part
        mov     ax, [bp+arg_0]
        cwd                    ; AX=low part d, DX=high part d
        sub     cx, ax
        mov     ax, cx
        sbb    bx, dx
        mov     dx, bx
        pop     bp
        retn    14
func3     endp

WinMain   proc near
        push    bp
        mov     bp, sp

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov     ax, 123
push   ax
mov     ax, 456
push   ax
mov     ax, 789
push   ax
call   func1
mov     ax, 9      ; high part of 600000
push   ax
mov     ax, 27C0h ; low part of 600000
push   ax
mov     ax, 0Ah   ; high part of 700000
push   ax
mov     ax, 0AE60h ; low part of 700000
push   ax
mov     ax, 0Ch   ; high part of 800000
push   ax
mov     ax, 3500h ; low part of 800000
push   ax
call   func2
mov     ax, 9      ; high part of 600000
push   ax
mov     ax, 27C0h ; low part of 600000
push   ax
mov     ax, 0Ah   ; high part of 700000
push   ax
mov     ax, 0AE60h ; low part of 700000
push   ax
mov     ax, 0Ch   ; high part of 800000
push   ax
mov     ax, 3500h ; low part of 800000
push   ax
mov     ax, 7Bh   ; 123
push   ax
call   func3
xor     ax, ax    ; return 0
pop     bp
retn   0Ah
WinMain endp

```

32-битные значения (тип данных `long` означает 32-бита, а `int` здесь 16-битный) в 16-битном коде (и в MS-DOS и в Win16) передаются парами) . Это так же как и 64-битные значения передаются в 32-битной среде (?? (стр. ??)).

`sub_B2` здесь это библиотечная функция написанная разработчиками компилятора, делающая «long multiplication», т.е. перемножает два 32-битных значения. Другие функции компиляторов делающие то же самое перечислены здесь : ?? (стр. ??), ?? (стр. ??).

Пара инструкций `ADD/ADC` используется для сложения этих составных значений : `ADD` может установить или сбросить флаг `CF`, а `ADC` будет использовать его после.

Пара инструкций `SUB/SBB` используется для вычитания: `SUB` может установить

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

или сбросить флаг CF, SBB будет использовать его после.

32-битные значения возвращаются из функций в паре регистров DX:AX .

Константы так же передаются как пары в WinMain().

Константа 123 типа *int* в начале конвертируется (учитывая знак) в 32-битное значение используя инструкция CWD .

### 3.31.5. Пример #5

```
#include <windows.h>

int PASCAL string_compare (char *s1, char *s2)
{
    while (1)
    {
        if (*s1!=*s2)
            return 0;
        if (*s1==0 || *s2==0)
            return 1; // end of string
        s1++;
        s2++;
    };
};

int PASCAL string_compare_far (char far *s1, char far *s2)
{
    while (1)
    {
        if (*s1!=*s2)
            return 0;
        if (*s1==0 || *s2==0)
            return 1; // end of string
        s1++;
        s2++;
    };
};

void PASCAL remove_digits (char *s)
{
    while (*s)
    {
        if (*s>='0' && *s<='9')
            *s='-';
        s++;
    };
};

char str[]="hello 1234 world";

int PASCAL WinMain( HINSTANCE hInstance,
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        HINSTANCE hPrevInstance,
        LPSTR lpCmdLine,
        int nCmdShow )
{
    string_compare ("asd", "def");
    string_compare_far ("asd", "def");
    remove_digits (str);
    MessageBox (NULL, str, "caption", MB_YESNOCANCEL);
    return 0;
};

```

```

string_compare proc near

arg_0 = word ptr 4
arg_2 = word ptr 6

    push    bp
    mov     bp, sp
    push    si
    mov     si, [bp+arg_0]
    mov     bx, [bp+arg_2]

loc_12: ; CODE XREF: string_compare+21j
    mov     al, [bx]
    cmp     al, [si]
    jz      short loc_1C
    xor     ax, ax
    jmp     short loc_2B

loc_1C: ; CODE XREF: string_compare+Ej
    test    al, al
    jz      short loc_22
    jnz     short loc_27

loc_22: ; CODE XREF: string_compare+16j
    mov     ax, 1
    jmp     short loc_2B

loc_27: ; CODE XREF: string_compare+18j
    inc     bx
    inc     si
    jmp     short loc_12

loc_2B: ; CODE XREF: string_compare+12j
        ; string_compare+1Dj
    pop     si
    pop     bp
    retn    4
string_compare endp

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
string_compare_far proc near ; CODE XREF: WinMain+18p
```

```
arg_0 = word ptr 4
arg_2 = word ptr 6
arg_4 = word ptr 8
arg_6 = word ptr 0Ah
```

```
    push    bp
    mov     bp, sp
    push    si
    mov     si, [bp+arg_0]
    mov     bx, [bp+arg_4]
```

```
loc_3A: ; CODE XREF: string_compare_far+35j
```

```
    mov     es, [bp+arg_6]
    mov     al, es:[bx]
    mov     es, [bp+arg_2]
    cmp     al, es:[si]
    jz      short loc_4C
    xor     ax, ax
    jmp     short loc_67
```

```
loc_4C: ; CODE XREF: string_compare_far+16j
```

```
    mov     es, [bp+arg_6]
    cmp     byte ptr es:[bx], 0
    jz      short loc_5E
    mov     es, [bp+arg_2]
    cmp     byte ptr es:[si], 0
    jnz     short loc_63
```

```
loc_5E: ; CODE XREF: string_compare_far+23j
```

```
    mov     ax, 1
    jmp     short loc_67
```

```
loc_63: ; CODE XREF: string_compare_far+2Cj
```

```
    inc     bx
    inc     si
    jmp     short loc_3A
```

```
loc_67: ; CODE XREF: string_compare_far+1Aj
```

```
    ; string_compare_far+31j
```

```
    pop     si
    pop     bp
    retn    8
```

```
string_compare_far endp
```

```
remove_digits  proc near ; CODE XREF: WinMain+1Fp
```

```
arg_0 = word ptr 4
```

```

    push    bp
    mov     bp, sp
    mov     bx, [bp+arg_0]

loc_72: ; CODE XREF: remove_digits+18j
    mov     al, [bx]
    test    al, al
    jz      short loc_86
    cmp     al, 30h ; '0'
    jb     short loc_83
    cmp     al, 39h ; '9'
    ja     short loc_83
    mov     byte ptr [bx], 2Dh ; '-'

loc_83: ; CODE XREF: remove_digits+Ej
        ; remove_digits+12j
    inc     bx
    jmp     short loc_72

loc_86: ; CODE XREF: remove_digits+Aj
    pop     bp
    retn    2
remove_digits    endp

WinMain proc near ; CODE XREF: start+EDp
    push    bp
    mov     bp, sp
    mov     ax, offset aAsd ; "asd"
    push    ax
    mov     ax, offset aDef ; "def"
    push    ax
    call    string_compare
    push    ds
    mov     ax, offset aAsd ; "asd"
    push    ax
    push    ds
    mov     ax, offset aDef ; "def"
    push    ax
    call    string_compare_far
    mov     ax, offset aHello1234World ; "hello 1234 world"
    push    ax
    call    remove_digits
    xor     ax, ax
    push    ax
    push    ds
    mov     ax, offset aHello1234World ; "hello 1234 world"
    push    ax
    push    ds
    mov     ax, offset aCaption ; "caption"
    push    ax
    mov     ax, 3 ; MB_YESNOCANCEL
    push    ax

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

    call    MESSAGEBOX
    xor     ax, ax
    pop     bp
    retn    0Ah
WinMain endp

```

Здесь мы можем увидеть разницу между указателями «near» и указателями «far» еще один ужасный артефакт сегментированной памяти 16-битного 8086 .

Читайте больше об этом: ?? (стр. ??).

Указатели «near» («близкие») это те которые указывают в пределах текущего сегмента . Поэтому, функция `string_compare()` берет на вход только 2 16-битных значения и работает с данными расположенными в сегменте, на который указывает DS (инструкция `mov al, [bx]` на самом деле работает как `mov al, ds:[bx]` — DS используется здесь неявно).

Указатели «far» (далекие) могут указывать на данные в другом сегменте памяти.

Поэтому `string_compare_far()` берет на вход 16-битную пару как указатель, загружает старшую часть в сегментный регистр ES и обращается к данным через него

(`mov al, es:[bx]`). Указатели «far» также используются в моем win16-примере касательно

`MessageBox(): ??` (стр. ??). Действительно, ядро Windows должно знать, из какого сегмента данных читать текстовые строки, так что ему нужна полная информация.

Причина этой разницы в том, что компактная программа вполне может обойтись одним сегментом данных размером 64 килобайта, так что старшую часть указателя передавать не нужна (ведь она одинаковая везде) . Большие программы могут использовать несколько сегментов данных размером 64 килобайта, так что нужно указывать каждый раз, в каком сегменте расположены данные .

То же касается и сегментов кода. Компактная программа может расположиться в пределах одного 64kb-сегмента, тогда функции в ней будут вызываться инструкцией CALL NEAR, а возвращаться управление используя RETN. Но если сегментов кода несколько, тогда и адрес вызываемой функции будет задаваться парой, вызываться она будет используя CALL FAR, а возвращаться управление используя RETF.

Это то что задается в компиляторе указывая «memory model».

Компиляторы под MS-DOS и Win16 имели разные библиотеки под разные модели памяти: они отличались типами указателей для кода и данных.

### 3.31.6. Пример #6

```

#include <windows.h>
#include <time.h>
#include <stdio.h>

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

char strbuf[256];

int PASCAL WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow )
{
    struct tm *t;
    time_t unix_time;

    unix_time=time(NULL);

    t=localtime (&unix_time);

    sprintf (strbuf, "%04d-%02d-%02d %02d:%02d:%02d", t->tm_year+1900, ↵
    ↵ t->tm_mon, t->tm_mday,
        t->tm_hour, t->tm_min, t->tm_sec);

    MessageBox (NULL, strbuf, "caption", MB_OK);
    return 0;
};

```

```

WinMain      proc near

var_4        = word ptr -4
var_2        = word ptr -2

    push     bp
    mov     bp, sp
    push     ax
    push     ax
    xor     ax, ax
    call    time_
    mov     [bp+var_4], ax    ; low part of UNIX time
    mov     [bp+var_2], dx    ; high part of UNIX time
    lea     ax, [bp+var_4]    ; take a pointer of high part
    call    localtime_
    mov     bx, ax            ; t
    push    word ptr [bx]    ; second
    push    word ptr [bx+2]  ; minute
    push    word ptr [bx+4]  ; hour
    push    word ptr [bx+6]  ; day
    push    word ptr [bx+8]  ; month
    mov     ax, [bx+0Ah]     ; year
    add     ax, 1900
    push    ax
    mov     ax, offset a04d02d02d02d02 ;
"%04d-%02d-%02d %02d:%02d:%02d"
    push    ax
    mov     ax, offset strbuf
    push    ax
    call    sprintf_

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        add     sp, 10h
        xor     ax, ax           ; NULL
        push   ax
        push   ds
        mov    ax, offset strbuf
        push   ax
        push   ds
        mov    ax, offset aCaption ; "caption"
        push   ax
        xor     ax, ax           ; MB_OK
        push   ax
        call   MESSAGEBOX
        xor     ax, ax
        mov    sp, bp
        pop    bp
        retn   0Ah
WinMain endp

```

Время в формате UNIX это 32-битное значение, так что оно возвращается в паре регистров DX:AX и сохраняется в двух локальных 16-битных переменных. Потом указатель на эту пару передается в функцию `localtime()`. Функция `localtime()` имеет структуру `struct tm` расположенную у себя где-то внутри, так что только указатель на нее возвращается. Кстати, это также означает, что функцию нельзя вызывать еще раз, пока её результаты не были использованы.

Для функций `time()` и `localtime()` используется Watcom-соглашение о вызовах: первые четыре аргумента передаются через регистры AX, DX, BX и CX, а остальные аргументы через стек. Функции, использующие это соглашение, маркируется символом подчеркивания в конце имени.

Для вызова функции `sprintf()` используется обычное соглашение *cdecl* (?? (стр. ??)) вместо PASCAL или Watcom, так что аргументы передаются привычным образом.

## Глобальные переменные

Это тот же пример, только переменные теперь глобальные:

```

#include <windows.h>
#include <time.h>
#include <stdio.h>

char strbuf[256];
struct tm *t;
time_t unix_time;

int PASCAL WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow )
{

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    unix_time=time(NULL);

    t=localtime (&unix_time);

    sprintf (strbuf, "%04d-%02d-%02d %02d:%02d:%02d", t->tm_year+1900, ↗
↘ t->tm_mon, t->tm_mday,
        t->tm_hour, t->tm_min, t->tm_sec);

    MessageBox (NULL, strbuf, "caption", MB_OK);
    return 0;
};

```

```

unix_time_low    dw 0
unix_time_high   dw 0
t                dw 0

WinMain          proc near
    push        bp
    mov         bp, sp
    xor         ax, ax
    call        time_
    mov         unix_time_low, ax
    mov         unix_time_high, dx
    mov         ax, offset unix_time_low
    call        localtime_
    mov         bx, ax
    mov         t, ax                ; will not be used in future...
    push        word ptr [bx]        ; seconds
    push        word ptr [bx+2]      ; minutes
    push        word ptr [bx+4]      ; hour
    push        word ptr [bx+6]      ; day
    push        word ptr [bx+8]      ; month
    mov         ax, [bx+0Ah]         ; year
    add         ax, 1900
    push        ax
    mov         ax, offset a04d02d02d02d02 ;
    "%04d-%02d-%02d %02d:%02d:%02d"
    push        ax
    mov         ax, offset strbuf
    push        ax
    call        sprintf_
    add         sp, 10h
    xor         ax, ax                ; NULL
    push        ax
    push        ds
    mov         ax, offset strbuf
    push        ax
    push        ds
    mov         ax, offset aCaption ; "caption"
    push        ax
    xor         ax, ax                ; MB_OK
    push        ax
    call        MESSAGEBOX
    xor         ax, ax                ; return 0

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

WinMain	pop retn endp	bp 0Ah
---------	---------------------	-----------

t не будет использоваться, но компилятор создал код, записывающий в эту переменную.

Потому что он не уверен, может быть это значение будет прочитано где-то в другом модуле.

# Глава 4

## Java

### 4.1. Java

#### 4.1.1. Введение

Есть немало известных декомпиляторов для Java (или для **JVM!**<sup>1</sup>-байткода вообще)<sup>2</sup>.

Причина в том что декомпиляция **JVM!**-байткода проще чем низкоуровневого x86-кода:

- Здесь намного больше информации о типах.
- Модель памяти в **JVM!** более строгая и очерченная.
- Java-компилятор не делает никаких оптимизаций (это делает **JVM! JIT!**<sup>3</sup> во время исполнения), так что байткод в class-файлах легко читаем.

Когда знания **JVM!**-байткода могут быть полезны?

- Мелкая/несложная работа по патчингу class-файлов без необходимости снова компилировать результаты декомпилятора.
- Анализ обфусцированного кода.
- Анализ кода сгенерированного более новым Java-компилятором, для которого еще пока нет обновленного декомпилятора.
- Создание вашего собственного обфускатора.
- Создание кодегенератора компилятора (back-end), создающего код для **JVM!** (как Scala, Clojure, и т. д.<sup>4</sup>).

Начнем с простых фрагментов кода.

---

<sup>1</sup>**JVM!**

<sup>2</sup>Например, JAD: <http://varanekas.com/jad/>

<sup>3</sup>**JIT!**

<sup>4</sup>Полный список: [http://en.wikipedia.org/wiki/List\\_of\\_JVM\\_languages](http://en.wikipedia.org/wiki/List_of_JVM_languages)

Если не указано иное, везде используется JDK 1.7.

Эта команда использовалась везде для декомпиляции class-файлов:  
javap -c -verbose.

Эта книга использовалась мною для подготовки всех примеров: [Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, *The Java(R) Virtual Machine Specification / Java SE 7 Edition*] <sup>5</sup>.

#### 4.1.2. Возврат значения

Наверное, самая простая из всех возможных функций на Java это та, что возвращает некоторое значение.

О, и мы не должны забывать, что в Java нет «свободных» функций в общем смысле, это «методы».

Каждый метод принадлежит какому-то классу, так что невозможно объявить метод вне какого-либо класса.

Но мы все равно будем называть их «функциями», для простоты.

```
public class ret
{
    public static int main(String[] args)
    {
        return 0;
    }
}
```

Компилируем это:

```
javac ret.java
```

...и декомпилирую используя стандартную утилиту в Java:

```
javap -c -verbose ret.class
```

И получаем:

Листинг 4.1: JDK 1.7 (excerpt)

```
public static int main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
     0: iconst_0
     1: ireturn
```

<sup>5</sup>Также доступно здесь: <https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf>; <http://docs.oracle.com/javase/specs/jvms/se7/html/>

Разработчики Java решили, что 0 это самая используемая константа в программировании, так что здесь есть отдельная однобайтная инструкция `iconst_0`, заталкивающая 0 в стек <sup>6</sup>.

Здесь есть также `iconst_1` (заталкивающая 1), `iconst_2`, и т. д., вплоть до `iconst_5`. Есть также `iconst_m1` заталкивающая -1.

Стек также используется в JVM для передачи данных в вызывающие ф-ции, и также для возврата значений. Так что `iconst_0` заталкивает 0 в стек. `ireturn` возвращает целочисленное значение (*i* в названии означает *integer*) из **TOS!**<sup>7</sup>.

Немного перепишем наш пример, теперь возвращаем 1234:

```
public class ret
{
    public static int main(String[] args)
    {
        return 1234;
    }
}
```

...получаем:

Листинг 4.2: JDK 1.7 (excerpt)

```
public static int main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
     0: sipush      1234
     3: ireturn
```

`sipush` (*short integer*) заталкивает значение 1234 в стек. *short* в имени означает, что 16-битное значение будет заталкиваться в стек.

Число 1234 действительно помещается в 16-битное значение.

Как насчет больших значений?

```
public class ret
{
    public static int main(String[] args)
    {
        return 12345678;
    }
}
```

Листинг 4.3: Constant pool

```
...
#2 = Integer      12345678
...
```

<sup>6</sup>Так же как и в MIPS, где для нулевой константы имеется отдельный регистр: ?? (стр. ??).

<sup>7</sup>**TOS!**



```

public static int main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
     0: ldc          #2          // int 12345678
     2: ireturn

```

Невозможно закодировать 32-битное число в опкоде какой-либо JVM-инструкции, разработчики не оставили такой возможности.

Так что 32-битное число 12345678 сохранено в так называемом «constant pool» (пул констант), который, так скажем, является библиотекой наиболее используемых констант (включая строки, объекты, и т. д.).

Этот способ передачи констант не уникален для JVM.

MIPS, ARM и прочие RISC-процессоры не могут кодировать 32-битные числа в 32-битных опкодах, так что код для RISC-процессоров (включая MIPS и ARM) должен конструировать значения в несколько шагов, или держать их в сегменте данных: ?? (стр. ??), ?? (стр. ??).

Код для MIPS также традиционно имеет пул констант, называемый «literal pool», это сегменты с названиями «.lit4» (для хранения 32-битных чисел с плавающей точкой одинарной точности) и «.lit8» (для хранения 64-битных чисел с плавающей точкой двойной точности).

Попробуем некоторые другие типы данных!

Boolean:

```

public class ret
{
    public static boolean main(String[] args)
    {
        return true;
    }
}

```

```

public static boolean main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
     0: iconst_1
     1: ireturn

```

Этот JVM-байткод не отличается от того, что возвращает целочисленную 1.

32-битные слоты данных в стеке также используются для булевых значений, как в Си/Си++.

Но нельзя использовать возвращаемое значение булевого типа как целочисленное и наоборот — информация о типах сохраняется в class-файлах и проверяется при запуске.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Та же история с 16-битным *short*:

```
public class ret
{
    public static short main(String[] args)
    {
        return 1234;
    }
}
```

```
public static short main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=1, locals=1, args_size=1
     0: sipush      1234
     3: ireturn
```

...и *char*!

```
public class ret
{
    public static char main(String[] args)
    {
        return 'A';
    }
}
```

```
public static char main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=1, locals=1, args_size=1
     0: bipush      65
     2: ireturn
```

*bipush* означает «push byte».

Нужно сказать, что *char* в Java, это 16-битный символ в кодировке UTF-16, и он эквивалентен *short*, но ASCII-код символа «A» это 65, и можно воспользоваться инструкцией для передачи байта в стек.

Попробуем также *byte*:

```
public class retc
{
    public static byte main(String[] args)
    {
        return 123;
    }
}
```

```
public static byte main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

stack=1, locals=1, args_size=1
 0: bipush      123
 2: ireturn

```

Кто-то может спросить, зачем заморачиваться использованием 16-битного типа *short*, который внутри все равно 32-битный *integer*?

Зачем использовать тип данных *char*, если это то же самое что и тип *short*?

Ответ прост: для контроля типов данных и читабельности исходников.

*char* может быть эквивалентом *short*, но мы быстро понимаем, что это ячейка для символа в кодировке UTF-16, а не для какого-то другого целочисленного значения.

Когда используем *short*, мы можем показать всем, что диапазон этой переменной ограничен 16-ю битами.

Очень хорошая идея использовать тип *boolean* где нужно, вместо *int* для тех же целей, как это было в Си.

В Java есть также 64-битный целочисленный тип:

```

public class ret3
{
    public static long main(String[] args)
    {
        return 1234567890123456789L;
    }
}

```

Листинг 4.4: Constant pool

```

...
#2 = Long      1234567890123456789L
...

```

```

public static long main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=1, args_size=1
     0: ldc2_w      #2          // long 1234567890123456789L
     3: lreturn

```

64-битное число также хранится в пуле констант, *ldc2\_w* загружает его и *lreturn* (*long return*) возвращает его.

Инструкция *ldc2\_w* также используется для загрузки чисел с плавающей точкой двойной точности (которые также занимают 64 бита) из пула констант:

```

public class ret
{
    public static double main(String[] args)
    {
        return 123.456d;
    }
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
}
}
```

Листинг 4.5: Constant pool

```
...
#2 = Double          123.456d
...
```

```
public static double main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=1, args_size=1
     0: ldc2_w          #2              // double 123.456d
     3: dreturn
```

dreturn означает «return double».

И наконец, числа с плавающей точкой одинарной точности:

```
public class ret
{
    public static float main(String[] args)
    {
        return 123.456f;
    }
}
```

Листинг 4.6: Constant pool

```
...
#2 = Float           123.456f
...
```

```
public static float main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
     0: ldc              #2              // float 123.456f
     2: freturn
```

Используемая здесь инструкция ldc та же, что и для загрузки 32-битных целочисленных чисел из пула констант.

freturn означает «return float».

А что насчет тех случаев, когда функция ничего не возвращает?

```
public class ret
{
    public static void main(String[] args)
    {
        return;
    }
}
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=0, locals=1, args_size=1
     0: return
```

Это означает, что инструкция `return` используется для возврата управления без возврата какого-либо значения.

Зная все это, по последней инструкции очень легко определить тип возвращаемого значения функции (или метода).

### 4.1.3. Простая вычисляющая функция

Продолжим с простой вычисляющей функцией.

```
public class calc
{
    public static int half(int a)
    {
        return a/2;
    }
}
```

Это тот случай, когда используется инструкция `iconst_2`:

```
public static int half(int);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=1, args_size=1
     0: iload_0
     1: iconst_2
     2: idiv
     3: ireturn
```

`iload_0` берет нулевой аргумент функции и заталкивает его в стек. `iconst_2` заталкивает в стек 2.

Вот как выглядит стек после исполнения этих двух инструкций:

```
      +----+
TOS ->| 2 |
      +----+
      | a |
      +----+
```

`idiv` просто берет два значения на вершине стека (**TOS!**), делит одно на другое и оставляет результат на вершине (**TOS!**):

```
      +-----+
TOS ->| result |
      +-----+
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

`ireturn` берет его и возвращает.

Продолжим с числами с плавающей запятой, двойной точности:

```
public class calc
{
    public static double half_double(double a)
    {
        return a/2.0;
    }
}
```

Листинг 4.7: Constant pool

```
...
#2 = Double          2.0d
...
```

```
public static double half_double(double);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=4, locals=2, args_size=1
   0: dload_0
   1: ldc2_w      #2          // double 2.0d
   4: ddiv
   5: dreturn
```

Почти то же самое, но инструкция `ldc2_w` используется для загрузки константы 2.0 из пула констант.

Также, все три инструкции имеют префикс *d*, что означает, что они работают с переменными типа *double*.

Теперь перейдем к функции с двумя аргументами:

```
public class calc
{
    public static int sum(int a, int b)
    {
        return a+b;
    }
}
```

```
public static int sum(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=2, locals=2, args_size=2
   0: iload_0
   1: iload_1
   2: iadd
   3: ireturn
```

`iload_0` загружает первый аргумент функции (a), `iload_1` — второй (b).

Вот так выглядит стек после исполнения обеих инструкций:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

+----+
TOS ->| b |
+----+
| a |
+----+

```

`iadd` складывает два значения и оставляет результат на **TOS!**:

```

+-----+
TOS ->| result |
+-----+

```

Расширим этот пример до типа данных *long*:

```

public static long lsum(long a, long b)
{
    return a+b;
}

```

...получим:

```

public static long lsum(long, long);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=4, locals=4, args_size=2
   0: lload_0
   1: lload_2
   2: ladd
   3: lreturn

```

Вторая инструкция `lload` берет второй аргумент из второго слота.

Это потому что 64-битное значение *long* занимает ровно два 32-битных слота.

Немного более сложный пример:

```

public class calc
{
    public static int mult_add(int a, int b, int c)
    {
        return a*b+c;
    }
}

```

```

public static int mult_add(int, int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=2, locals=3, args_size=3
   0: iload_0
   1: iload_1
   2: imul
   3: iload_2
   4: iadd
   5: ireturn

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Первый шаг это умножение. Произведение остается на **TOS!**:

```

+-----+
TOS ->| product |
+-----+

```

`iload_2` загружает третий аргумент (с) в стек:

```

+-----+
TOS ->|   c   |
+-----+
      | product |
+-----+

```

Теперь инструкция `iadd` может сложить два значения.

#### 4.1.4. Модель памяти в JVM!

x86 и другие низкоуровневые среды используют стек для передачи аргументов и как хранилище локальных переменных. **JVM!** устроена немного иначе.

Тут есть:

- Массив локальных переменных (**LVA!**<sup>8</sup>).

Используется как хранилище для аргументов функций и локальных переменных.

Инструкции вроде `iload_0` загружают значения оттуда. `istore` записывает значения туда.

В начале идут аргументы функции: начиная с 0, или с 1 (если нулевой аргумент занят указателем *this*).

Затем располагаются локальные переменные.

Каждый слот имеет размер 32 бита.

Следовательно, значения типов *long* и *double* занимают два слота.

- Стек операндов (или просто «стек»).

Используется для вычислений и для передачи аргументов во время вызова других функций.

В отличие от низкоуровневых сред вроде x86, здесь невозможно работать со стеком без использования инструкций, которые явно заталкивают или выталкивают значения туда/оттуда.

- Куча (heap). Используется как хранилище для объектов и массивов.

Эти 3 области изолированы друг от друга.

<sup>8</sup>LVA!



### 4.1.5. Простой вызов функций

`Math.random()` возвращает псевдослучайное число в пределах `[0.0 ...1.0)`, но представим, по какой-то причине, нам нужна функция, возвращающая число в пределах `[0.0 ...0.5)`:

```
public class HalfRandom
{
    public static double f()
    {
        return Math.random()/2;
    }
}
```

Листинг 4.8: Constant pool

```
...
#2 = Methodref      #18.#19    // java/lang/Math.random:()D
#3 = Double         2.0d
...
#12 = Utf8          ()D
...
#18 = Class         #22         // java/lang/Math
#19 = NameAndType   #23:#12    // random:()D
#22 = Utf8          java/lang/Math
#23 = Utf8          random
```

```
public static double f();
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=4, locals=0, args_size=0
     0: invokestatic #2          // Method java/lang/Math.random:()D
     3: ldc2_w        #3          // double 2.0d
     6: ddiv
     7: dreturn
```

`invokestatic` вызывает функцию `Math.random()` и оставляет результат на **TOS!**. Затем результат делится на `2.0` и возвращается.

Но как закодировано имя функции?

Оно закодировано в пуле констант используя выражение `Methodref`.

Оно определяет имена класса и метода.

Первое поле `Methodref` указывает на выражение `Class`, которое, в свою очередь, указывает на обычную текстовую строку («`java/lang/Math`»).

Второе выражение `Methodref` указывает на выражение `NameAndType`, которое также имеет две ссылки на строки.

Первая строка это «`random`», это имя метода.

Вторая строка это «`()D`», которая кодирует тип функции. Это означает, что возвращаемый тип — `double` (отсюда `D` в строке).

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Благодаря этому 1) JVM проверяет корректность типов данных; 2) Java-декомпиляторы могут восстанавливать типы данных из class-файлов.

Наконец попробуем пример «Hello, world!»:

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World");
    }
}
```

Листинг 4.9: Constant pool

```
...
#2 = Fieldref      #16.#17      // java/lang/System.out:Ljava/io/
↳ /PrintStream;
#3 = String        #18          // Hello, World
#4 = Methodref     #19.#20      // java/io/PrintStream.println:(
↳ Ljava/lang/String;)V
...
#16 = Class        #23          // java/lang/System
#17 = NameAndType  #24:#25      // out:Ljava/io/PrintStream;
#18 = Utf8         Hello, World
#19 = Class        #26          // java/io/PrintStream
#20 = NameAndType  #27:#28      // println:(Ljava/lang/String;)V
...
#23 = Utf8         java/lang/System
#24 = Utf8         out
#25 = Utf8         Ljava/io/PrintStream;
#26 = Utf8         java/io/PrintStream
#27 = Utf8         println
#28 = Utf8         (Ljava/lang/String;)V
...
```

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=1, args_size=1
     0: getstatic   #2          // Field java/lang/System.out:Ljava/io/
↳ PrintStream;
     3: ldc         #3          // String Hello, World
     5: invokevirtual #4          // Method java/io/PrintStream.println:(
↳ Ljava/lang/String;)V
     8: return
```

ldc по смещению 3 берет указатель (или адрес) на строку «Hello, World» в пуле констант и заталкивает его в стек.

В мире Java это называется *reference*, но это скорее указатель или просто адрес <sup>9</sup>.

<sup>9</sup>О разнице между указателями и *reference* в C++: ?? (стр. ??).

Уже знакомая нам инструкция `invokevirtual` берет информацию о функции (или методе) `println` из пула констант и вызывает её.

Как мы можем знать, есть несколько методов `println`, каждый предназначен для каждого типа данных.

В нашем случае, используется та версия `println`, которая для типа данных `String`.

Что насчет первой инструкции `getstatic`?

Эта инструкция берет *reference* (или адрес) поля объекта `System.out` и заталкивает его в стек.

Это значение работает как указатель *this* для метода `println`.

Таким образом, внутри, метод `println` берет на вход два аргумента: 1) *this*, т.е. указатель на объект <sup>10</sup>; 2) адрес строки «Hello, World».

Действительно, `println()` вызывается как метод в рамках инициализированного объекта `System.out`.

Для удобства, утилита `javap` пишет всю эту информацию в комментариях.

#### 4.1.6. Вызов `beep()`

Вот простейший вызов двух функций без аргументов:

```
public static void main(String[] args)
{
    java.awt.Toolkit.getDefaultToolkit().beep();
};
```

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
     0: invokestatic #2      // Method java/awt/Toolkit.↵
    ↵ getDefaultToolkit():Ljava/awt/Toolkit;
     3: invokevirtual #3      // Method java/awt/Toolkit.beep:()V
     6: return
```

Первая `invokestatic` по смещению 0 вызывает `java.awt.Toolkit.getDefaultToolkit()`, которая возвращает *reference* (указатель) на объект класса `Toolkit`.

Инструкция `invokevirtual` по смещению 3 вызывает метод `beep()` этого класса.

<sup>10</sup>Или «экземпляр класса» в некоторой русскоязычной литературе.

### 4.1.7. Линейный конгруэнтный PRNG!

Попробуем простой генератор псевдослучайных чисел, который мы однажды уже рассматривали в этой книге (?? (стр. ??)):

```
public class LCG
{
    public static int rand_state;

    public void my_srand (int init)
    {
        rand_state=init;
    }

    public static int RNG_a=1664525;
    public static int RNG_c=1013904223;

    public int my_rand ()
    {
        rand_state=rand_state*RNG_a;
        rand_state=rand_state+RNG_c;
        return rand_state & 0x7fff;
    }
}
```

Здесь пара полей класса, которые инициализируются в начале. Но как?

В выводе javap мы можем найти конструктор класса:

```
static {};  
  flags: ACC_STATIC  
  Code:  
    stack=1, locals=0, args_size=0  
     0: ldc          #5          // int 1664525  
     2: putstatic     #3          // Field RNG_a:I  
     5: ldc          #6          // int 1013904223  
     7: putstatic     #4          // Field RNG_c:I  
    10: return
```

Так инициализируются переменные.

RNG\_a занимает третий слот в классе и RNG\_c — четвертый, и putstatic записывает туда константы.

Функция my\_srand() просто записывает входное значение в rand\_state:

```
public void my_srand(int);  
  flags: ACC_PUBLIC  
  Code:  
    stack=1, locals=2, args_size=2  
     0: iload_1  
     1: putstatic     #2          // Field rand_state:I  
     4: return
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

`iload_1` берет входное значение и заталкивает его в стек. Но почему не `iload_0`?

Это потому что эта функция может использовать поля класса, а переменная *this* также передается в эту функцию как нулевой аргумент.

Поле `rand_state` занимает второй слот в классе, так что `putstatic` копирует переменную из **TOS!** во второй слот.

Теперь `my_rand()`:

```
public int my_rand();
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=1, args_size=1
     0: getstatic    #2      // Field rand_state:I
     3: getstatic    #3      // Field RNG_a:I
     6: imul
     7: putstatic    #2      // Field rand_state:I
    10: getstatic    #2      // Field rand_state:I
    13: getstatic    #4      // Field RNG_c:I
    16: iadd
    17: putstatic    #2      // Field rand_state:I
    20: getstatic    #2      // Field rand_state:I
    23: sipush      32767
    26: iand
    27: ireturn
```

Она просто загружает все переменные из полей объекта, производит с ними операции и обновляет значение `rand_state`, используя инструкцию `putstatic`.

По смещению 20, значение `rand_state` перезагружается снова (это потому что оно было выброшено из стека перед этим, инструкцией `putstatic`).

Это выглядит как неэффективный код, но можете быть уверенными, **JVM!** обычно достаточно хорош, чтобы хорошо оптимизировать подобные вещи.

#### 4.1.8. Условные переходы

Перейдем к условным переходам.

```
public class abs
{
    public static int abs(int a)
    {
        if (a<0)
            return -a;
        return a;
    }
}
```

```
public static int abs(int);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

0: iload_0
1: ifge      7
4: iload_0
5: ineg
6: ireturn
7: iload_0
8: ireturn

```

`ifge` переходит на смещение 7 если значение на **TOS!** больше или равно 0.

Не забывайте, любая инструкция `ifXX` выталкивает значение (с которым будет производиться сравнение) из стека.

`ineg` просто меняет знак значения на **TOS!**.

Еще пример:

```

public static int min (int a, int b)
{
    if (a>b)
        return b;
    return a;
}

```

Получаем:

```

public static int min(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=2, locals=2, args_size=2
  0: iload_0
  1: iload_1
  2: if_icmple    7
  5: iload_1
  6: ireturn
  7: iload_0
  8: ireturn

```

`if_icmple` выталкивает два значения и сравнивает их.

Если второе меньше первого (или равно), происходит переход на смещение 7.

Когда мы определяем функцию `max()` ...

```

public static int max (int a, int b)
{
    if (a>b)
        return a;
    return b;
}

```

...итоговый код точно такой же, только последние инструкции `iload` (на смещениях 5 и 7) поменяны местами:

```

public static int max(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=2, locals=2, args_size=2
   0: iload_0
   1: iload_1
   2: if_icmple      7
   5: iload_0
   6: ireturn
   7: iload_1
   8: ireturn

```

Более сложный пример:

```

public class cond
{
    public static void f(int i)
    {
        if (i<100)
            System.out.print("<100");
        if (i==100)
            System.out.print("==100");
        if (i>100)
            System.out.print(">100");
        if (i==0)
            System.out.print("==0");
    }
}

```

```

public static void f(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=2, locals=1, args_size=1
   0: iload_0
   1: bipush      100
   3: if_icmpge   14
   6: getstatic   #2          // Field java/lang/System.out:Ljava/io/
↳ PrintStream;
   9: ldc         #3          // String <100
  11: invokevirtual #4          // Method java/io/PrintStream.print:(
↳ Ljava/lang/String;)V
  14: iload_0
  15: bipush      100
  17: if_icmpne   28
  20: getstatic   #2          // Field java/lang/System.out:Ljava/io/
↳ PrintStream;
  23: ldc         #5          // String ==100
  25: invokevirtual #4          // Method java/io/PrintStream.print:(
↳ Ljava/lang/String;)V
  28: iload_0
  29: bipush      100
  31: if_icmple   42

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

34: getstatic    #2      // Field java/lang/System.out:Ljava/io/PrintStream;
↳ PrintStream;
37: ldc          #6      // String >100
39: invokevirtual #4      // Method java/io/PrintStream.print:(Ljava/lang/String;)V
↳ Ljava/lang/String;)V
42: iload_0
43: ifne         54
46: getstatic    #2      // Field java/lang/System.out:Ljava/io/PrintStream;
↳ PrintStream;
49: ldc          #7      // String ==0
51: invokevirtual #4      // Method java/io/PrintStream.print:(Ljava/lang/String;)V
↳ Ljava/lang/String;)V
54: return

```

`if_icmpge` Вытаскивает два значения и сравнивает их.

Если второй больше первого, или равен первому, происходит переход на смещение 14.

`if_icmpne` и `if_icmple` работают одинаково, но используются разные условия.

По смещению 43 есть также инструкция `ifne`.

Название неудачное, её было бы лучше назвать `ifnz` (переход если переменная на **TOS!** не равна нулю).

И вот что она делает: производит переход на смещение 54, если входное значение не ноль.

Если ноль, управление передается на смещение 46, где выводится строка «==0».

N.B.: В **JVM!** нет беззнаковых типов данных, так что инструкции сравнения работают только со знаковыми целочисленными значениями.

#### 4.1.9. Передача аргументов

Теперь расширим пример `min()/max()`:

```

public class minmax
{
    public static int min (int a, int b)
    {
        if (a>b)
            return b;
        return a;
    }

    public static int max (int a, int b)
    {
        if (a>b)
            return a;
        return b;
    }

    public static void main(String[] args)

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

    {
        int a=123, b=456;
        int max_value=max(a, b);
        int min_value=min(a, b);
        System.out.println(min_value);
        System.out.println(max_value);
    }
}

```

Вот код функции main():

```

public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=5, args_size=1
     0: bipush      123
     2: istore_1
     3: sipush      456
     6: istore_2
     7: iload_1
     8: iload_2
     9: invokestatic #2      // Method max:(II)I
    12: istore_3
    13: iload_1
    14: iload_2
    15: invokestatic #3      // Method min:(II)I
    18: istore     4
    20: getstatic   #4      // Field java/lang/System.out:Ljava/io/
    ↪ PrintStream;
    23: iload      4
    25: invokevirtual #5      // Method java/io/PrintStream.println:(I
    ↪ )V
    28: getstatic   #4      // Field java/lang/System.out:Ljava/io/
    ↪ PrintStream;
    31: iload_3
    32: invokevirtual #5      // Method java/io/PrintStream.println:(I
    ↪ )V
    35: return

```

В другую функцию аргументы передаются в стеке, а возвращаемое значение остается на **TOS!**.

#### 4.1.10. Битовые поля

Все побитовые операции работают также, как и в любой другой **ISA!**:

```

public static int set (int a, int b)
{
    return a | 1<<b;
}

public static int clear (int a, int b)
{

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        return a & ~(1<<b));
    }

```

```

public static int set(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=3, locals=2, args_size=2
    0: iload_0
    1: iconst_1
    2: iload_1
    3: ishl
    4: ior
    5: ireturn

public static int clear(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=3, locals=2, args_size=2
    0: iload_0
    1: iconst_1
    2: iload_1
    3: ishl
    4: iconst_m1
    5: ixor
    6: iand
    7: ireturn

```

`iconst_m1` загружает `-1` в стек, это то же что и значение `0xFFFFFFFF`.

Операция XOR с `0xFFFFFFFF` в одном из операндов, это тот же эффект что инвертирование всех бит.

Попробуем также расширить все типы данных до 64-битного *long*:

```

public static long lset (long a, int b)
{
    return a | 1<<b;
}

public static long lclear (long a, int b)
{
    return a & ~(1<<b));
}

```

```

public static long lset(long, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=4, locals=3, args_size=2
    0: lload_0
    1: iconst_1
    2: iload_2
    3: ishl
    4: i2l

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

5: lor
6: lreturn

public static long lclear(long, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=4, locals=3, args_size=2
  0: lload_0
  1: iconst_1
  2: iload_2
  3: ishl
  4: iconst_m1
  5: ixor
  6: i2l
  7: land
  8: lreturn

```

Код такой же, но используются инструкции с префиксом *l*, которые работают с 64-битными значениями.

Так же, второй аргумент функции все еще имеет тип *int*, и когда 32-битное число в нем должно быть расширено до 64-битного значения, используется инструкция *i2l*, которая расширяет значение типа *integer* в значение типа *long*.

#### 4.1.11. Циклы

```

public class Loop
{
    public static void main(String[] args)
    {
        for (int i = 1; i <= 10; i++)
        {
            System.out.println(i);
        }
    }
}

```

```

public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=2, locals=2, args_size=1
  0: iconst_1
  1: istore_1
  2: iload_1
  3: bipush      10
  5: if_icmpgt   21
  8: getstatic   #2          // Field java/lang/System.out:Ljava/io/
↳ /PrintStream;
 11: iload_1
 12: invokevirtual #3          // Method java/io/PrintStream.println
↳ :(I)V
 15: iinc        1, 1

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
18: goto      2
21: return
```

`iconst_1` загружает 1 в **TOS!**, `istore_1` сохраняет её в первом слоте **LVA!**. Почему не нулевой слот?

Потому что функция `main()` имеет один аргумент (массив `String`), и указатель на него (или *reference*) сейчас в нулевом слоте.

Так что локальная переменная `i` всегда будет в первом слоте.

Инструкции по смещениями 3 и 5 сравнивают `i` с 10.

Если `i` больше, управление передается на смещение 21, где функция заканчивает работу.

Если нет, вызывается `println`.

`i` перезагружается по смещению 11, для `println`.

Кстати, мы вызываем метод `println` для типа данных *integer*, и мы видим это в комментариях: «(I)V» (*I* означает *integer* и *V* означает, что возвращаемое значение имеет тип *void*).

Когда `println` заканчивается, `i` увеличивается на 1 по смещению 15.

Первый операнд инструкции это номер слота (1), второй это число (1) для прибавления.

`goto` это просто GOTO, она переходит на начало цикла по смещению 2.

Перейдем к более сложному примеру:

```
public class Fibonacci
{
    public static void main(String[] args)
    {
        int limit = 20, f = 0, g = 1;

        for (int i = 1; i <= limit; i++)
        {
            f = f + g;
            g = f - g;
            System.out.println(f);
        }
    }
}
```

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=5, args_size=1
     0: bipush      20
     2: istore_1
     3: iconst_0
     4: istore_2
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

5: iconst_1
6: istore_3
7: iconst_1
8: istore      4
10: iload      4
12: iload_1
13: if_icmpgt  37
16: iload_2
17: iload_3
18: iadd
19: istore_2
20: iload_2
21: iload_3
22: isub
23: istore_3
24: getstatic  #2          // Field java/lang/System.out:Ljava/io/
↳ /PrintStream;
27: iload_2
28: invokevirtual #3        // Method java/io/PrintStream.println
↳ :(I)V
31: iinc       4, 1
34: goto      10
37: return

```

Вот карта слотов в **LVA!**:

- 0 — единственный аргумент функции main()
- 1 — *limit*, всегда содержит 20
- 2 — *f*
- 3 — *g*
- 4 — *i*

Мы видим, что компилятор Java расположил переменные в слотах **LVA!** в точно таком же порядке, в котором переменные были определены в исходном коде.

Существуют отдельные инструкции `istore` для слотов 0, 1, 2, 3, но не 4 и более, так что здесь есть `istore` с дополнительным операндом по смещению 8, которая имеет номер слота в операнде.

Та же история с `iload` по смещению 10.

Но не слишком ли это сомнительно, выделить целый слот для переменной *limit*, которая всегда содержит 20 (так что это по сути константа), и перезагружать её так часто?

**JIT!**-компилятор в **JVM!** обычно достаточно хорош, чтобы всё это оптимизировать.

Самостоятельное вмешательство в код, наверное, того не стоит.

### 4.1.12. switch()

Выражение switch() реализуется инструкцией tableswitch:

```
public static void f(int a)
{
    switch (a)
    {
        case 0: System.out.println("zero"); break;
        case 1: System.out.println("one\n"); break;
        case 2: System.out.println("two\n"); break;
        case 3: System.out.println("three\n"); break;
        case 4: System.out.println("four\n"); break;
        default: System.out.println("something unknown\n"); break;
    }
}
```

Проще не бывает:

```
public static void f(int);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=1, args_size=1
     0: iload_0
     1: tableswitch  { // 0 to 4
                   0: 36
                   1: 47
                   2: 58
                   3: 69
                   4: 80
                   default: 91
               }
    36: getstatic    #2      // Field java/lang/System.out:Ljava/io/
↳   PrintStream;
    39: ldc           #3      // String zero
    41: invokevirtual #4      // Method java/io/PrintStream.println:(
↳   Ljava/lang/String;)V
    44: goto         99
    47: getstatic    #2      // Field java/lang/System.out:Ljava/io/
↳   PrintStream;
    50: ldc           #5      // String one\n
    52: invokevirtual #4      // Method java/io/PrintStream.println:(
↳   Ljava/lang/String;)V
    55: goto         99
    58: getstatic    #2      // Field java/lang/System.out:Ljava/io/
↳   PrintStream;
    61: ldc           #6      // String two\n
    63: invokevirtual #4      // Method java/io/PrintStream.println:(
↳   Ljava/lang/String;)V
    66: goto         99
    69: getstatic    #2      // Field java/lang/System.out:Ljava/io/
↳   PrintStream;
    72: ldc           #7      // String three\n
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

74: invokevirtual #4 // Method java/io/PrintStream.println:(L
↳ Ljava/lang/String;)V
77: goto 99
80: getstatic #2 // Field java/lang/System.out:Ljava/io/
↳ PrintStream;
83: ldc #8 // String four\n
85: invokevirtual #4 // Method java/io/PrintStream.println:(L
↳ Ljava/lang/String;)V
88: goto 99
91: getstatic #2 // Field java/lang/System.out:Ljava/io/
↳ PrintStream;
94: ldc #9 // String something unknown\n
96: invokevirtual #4 // Method java/io/PrintStream.println:(L
↳ Ljava/lang/String;)V
99: return

```

### 4.1.13. Массивы

#### Простой пример

Создадим массив из 10-и чисел и заполним его:

```

public static void main(String[] args)
{
    int a[]=new int[10];
    for (int i=0; i<10; i++)
        a[i]=i;
    dump (a);
}

```

```

public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=3, args_size=1
0: bipush 10
2: newarray int
4: astore_1
5: iconst_0
6: istore_2
7: iload_2
8: bipush 10
10: if_icmpge 23
13: aload_1
14: iload_2
15: iload_2
16: iastore
17: iinc 2, 1
20: goto 7
23: aload_1
24: invokestatic #4 // Method dump:([I)V
27: return

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Инструкция `newarray` создает объект массива из 10 элементов типа `int`.

Размер массива выставляется инструкцией `ipush` и остается на **TOS!**

Тип массива выставляется в операнде инструкции `newarray`.

После исполнения `newarray`, *reference* (или указатель) только что созданного в куче (heap) массива остается на **TOS!**

`astore_1` сохраняет *reference* на него в первом слоте **LVA!**

Вторая часть функции `main()` это цикл, сохраняющий значение *i* в соответствующий элемент массива.

`aload_1` берет *reference* массива и сохраняет его в стеке.

`iastore` затем сохраняет значение из стека в массив, *reference* на который в это время находится на **TOS!**

Третья часть функции `main()` вызывает функцию `dump()`.

Аргумент для нее готовится инструкцией `aload_1` (смещение 23).

Перейдем к функции `dump()`:

```
public static void dump(int a[])
{
    for (int i=0; i<a.length; i++)
        System.out.println(a[i]);
}
```

```
public static void dump(int[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=3, locals=2, args_size=1
   0: iconst_0
   1: istore_1
   2: iload_1
   3: aload_0
   4: arraylength
   5: if_icmpge    23
   8: getstatic    #2      // Field java/lang/System.out:Ljava/io/
↳ PrintStream;
  11: aload_0
  12: iload_1
  13: iaload
  14: invokevirtual #3      // Method java/io/PrintStream.println:(I)
↳ V
  17: iinc         1, 1
  20: goto        2
  23: return
```

Входящий *reference* на массив в нулевом слоте.

Выражение `a.length` в исходном коде конвертируется в инструкцию `arraylength`, она берет *reference* на массив и оставляет размер массива на **TOS!**

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



Инструкция `iaload` по смещению 13 используется для загрузки элементов массива, она требует, чтобы в стеке присутствовал *reference* на массив (подготовленный `aload_0` на 11), а также индекс (подготовленный `iload_1` по смещению 12).

Нужно сказать, что инструкции с префиксом *a* могут быть неверно поняты, как инструкции работающие с массивами (*array*). Это неверно.

Эти инструкции работают с *reference*-ами на объекты.

А массивы и строки это тоже объекты.

### Суммирование элементов массива

Еще один пример:

```
public class ArraySum
{
    public static int f (int[] a)
    {
        int sum=0;
        for (int i=0; i<a.length; i++)
            sum=sum+a[i];
        return sum;
    }
}
```

```
public static int f(int[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=3, locals=3, args_size=1
     0: iconst_0
     1: istore_1
     2: iconst_0
     3: istore_2
     4: iload_2
     5: aload_0
     6: arraylength
     7: if_icmpge    22
    10: iload_1
    11: aload_0
    12: iload_2
    13: iaload
    14: iadd
    15: istore_1
    16: iinc         2, 1
    19: goto        4
    22: iload_1
    23: ireturn
```

Нулевой слот в **LVA!** содержит указатель (*reference*) на входной массив.

Первый слот **LVA!** содержит локальную переменную *sum*.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

### Единственный аргумент main() это также массив

Будем использовать единственный аргумент main(), который массив строк:

```
public class UseArgument
{
    public static void main(String[] args)
    {
        System.out.print("Hi, ");
        System.out.print(args[1]);
        System.out.println(". How are you?");
    }
}
```

Нулевой аргумент это имя программы (как в Си/Си++, и т. д.), так что первый аргумент это тот, что пользователь добавил первым.

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=1, args_size=1
     0: getstatic   #2      // Field java/lang/System.out:Ljava/io/
    ↪ PrintStream;
     3: ldc         #3      // String Hi,
     5: invokevirtual #4      // Method java/io/PrintStream.print:(
    ↪ Ljava/lang/String;)V
     8: getstatic   #2      // Field java/lang/System.out:Ljava/io/
    ↪ PrintStream;
    11: aload_0
    12: iconst_1
    13: aaload
    14: invokevirtual #4      // Method java/io/PrintStream.print:(
    ↪ Ljava/lang/String;)V
    17: getstatic   #2      // Field java/lang/System.out:Ljava/io/
    ↪ PrintStream;
    20: ldc         #5      // String . How are you?
    22: invokevirtual #6      // Method java/io/PrintStream.println:(
    ↪ Ljava/lang/String;)V
    25: return
```

aload\_0 на 11 загружают *reference* на нулевой слот **LVA!** (первый и единственный аргумент main()).

iconst\_1 и aaload на 12 и 13 берут *reference* на первый (считая с 0) элемент массива.

*Reference* на строковый объект на **TOS!** по смещению 14, и оттуда он берется методом println.

### Заранее инициализированный массив строк

```
class Month
{
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

public static String[] months =
{
    "January",
    "February",
    "March",
    "April",
    "May",
    "June",
    "July",
    "August",
    "September",
    "October",
    "November",
    "December"
};

public String get_month (int i)
{
    return months[i];
};
}

```

Функция `get_month()` проста:

```

public java.lang.String get_month(int);
flags: ACC_PUBLIC
Code:
  stack=2, locals=2, args_size=2
   0: getstatic    #2          // Field months:[Ljava/lang/String;
   3: iload_1
   4: aaload
   5: areturn

```

`aaload` работает с массивом *reference*-ов.

Строка в Java это объект, так что используются *a*-инструкции для работы с ними.

`areturn` возвращает *reference* на объект `String`.

Как инициализируется массив `months[]`?

```

static {};
flags: ACC_STATIC
Code:
  stack=4, locals=0, args_size=0
   0: bipush       12
   2: anewarray    #3          // class java/lang/String
   5: dup
   6: iconst_0
   7: ldc         #4          // String January
   9: aastore
  10: dup
  11: iconst_1

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

12: ldc          #5          // String February
14: aastore
15: dup
16: iconst_2
17: ldc          #6          // String March
19: aastore
20: dup
21: iconst_3
22: ldc          #7          // String April
24: aastore
25: dup
26: iconst_4
27: ldc          #8          // String May
29: aastore
30: dup
31: iconst_5
32: ldc          #9          // String June
34: aastore
35: dup
36: bipush      6
38: ldc          #10         // String July
40: aastore
41: dup
42: bipush      7
44: ldc          #11         // String August
46: aastore
47: dup
48: bipush      8
50: ldc          #12         // String September
52: aastore
53: dup
54: bipush      9
56: ldc          #13         // String October
58: aastore
59: dup
60: bipush     10
62: ldc          #14         // String November
64: aastore
65: dup
66: bipush     11
68: ldc          #15         // String December
70: aastore
71: putstatic   #2          // Field months:[Ljava/lang/String;
74: return

```

`anewarray` создает новый массив *reference*-ов (отсюда префикс *a*).

Тип объекта определяется в операнде `anewarray`, там текстовая строка «`java/lang/String`».

`bipush 12` перед `anewarray` устанавливает размер массива.

Новая для нас здесь инструкция: `dup`.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Это стандартная инструкция в стековых компьютерах (включая ЯП Forth), которая делает дубликат значения на **TOS!**.

Кстати, FPU 80x87 это тоже стековый компьютер, и в нем есть аналогичная инструкция – FDUP.

Она используется здесь для дублирования *reference*-а на массив, потому что инструкция *aastore* выталкивает из стека *reference* на массив, но последующая инструкция *aastore* снова нуждается в нем.

Компилятор Java решил, что лучше генерировать *dup* вместо генерации инструкции *getstatic* перед каждой операцией записи в массив (т.е. 11 раз).

*aastore* кладет *reference* (на строку) в массив по индексу взятому из **TOS!**.

И наконец, *putstatic* кладет *reference* на только что созданный массив во второе поле нашего объекта, т.е. в поле *months*.

### Функции с переменным кол-вом аргументов (variadic)

Функции с переменным кол-вом аргументов (variadic) на самом деле используют массивы:

```
public static void f(int... values)
{
    for (int i=0; i<values.length; i++)
        System.out.println(values[i]);
}

public static void main(String[] args)
{
    f (1,2,3,4,5);
}
```

```
public static void f(int...);
  flags: ACC_PUBLIC, ACC_STATIC, ACC_VARARGS
  Code:
    stack=3, locals=2, args_size=1
     0: iconst_0
     1: istore_1
     2: iload_1
     3: aload_0
     4: arraylength
     5: if_icmpge    23
     8: getstatic    #2          // Field java/lang/System.out:Ljava/io/
    ↙ PrintStream;
    11: aload_0
    12: iload_1
    13: iaload
    14: invokevirtual #3          // Method java/io/PrintStream.println:(I
    ↙ )V
    17: iinc         1, 1
    20: goto        2
    23: return
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

По смещению 3, f() просто берет массив переменных используя aload\_0.

Затем берет размер массива, и т. д.

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=4, locals=1, args_size=1
     0: iconst_5
     1: newarray      int
     3: dup
     4: iconst_0
     5: iconst_1
     6: iastore
     7: dup
     8: iconst_1
     9: iconst_2
    10: iastore
    11: dup
    12: iconst_2
    13: iconst_3
    14: iastore
    15: dup
    16: iconst_3
    17: iconst_4
    18: iastore
    19: dup
    20: iconst_4
    21: iconst_5
    22: iastore
    23: invokestatic #4      // Method f:([I)V
    26: return
```

Массив конструируется в main() используя инструкцию newarray, затем он заполняется, и вызывается f().

Кстати, объект массива не уничтожается в конце main().

В Java вообще нет деструкторов, потому что в JVM есть сборщик мусора (garbage collector), делающий это автоматически, когда считает нужным.

Как насчет метода format()?

Он берет на вход два аргумента: строку и массив объектов:

```
public PrintStream format(String format, Object... args)
```

( <http://docs.oracle.com/javase/tutorial/java/data/numberformat.html> )

Посмотрим:

```
public static void main(String[] args)
{
    int i=123;
    double d=123.456;
    System.out.format("int: %d double: %f.%n", i, d);
}
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

}

```

public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=7, locals=4, args_size=1
     0: bipush      123
     2: istore_1
     3: ldc2_w      #2          // double 123.456d
     6: dstore_2
     7: getstatic   #4          // Field java/lang/System.out:Ljava/io/
↳ /PrintStream;
    10: ldc        #5          // String int: %d double: %f.%n
    12: iconst_2
    13: anewarray   #6          // class java/lang/Object
    16: dup
    17: iconst_0
    18: iload_1
    19: invokestatic #7          // Method java/lang/Integer.valueOf:(I)
↳ )Ljava/lang/Integer;
    22: astore
    23: dup
    24: iconst_1
    25: dload_2
    26: invokestatic #8          // Method java/lang/Double.valueOf:(D)
↳ Ljava/lang/Double;
    29: astore
    30: invokevirtual #9         // Method java/io/PrintStream.format:(
↳ Ljava/lang/String;[Ljava/lang/Object;)Ljava/io/PrintStream;
    33: pop
    34: return

```

Так что в начале значения типов *int* и *double* конвертируются в объекты типов *Integer* и *Double* используя методы *valueOf*.

Метод *format()* требует на входе объекты типа *Object*, а так как классы *Integer* и *Double* наследуются от корневого класса *Object*, они подходят как элементы во входном массиве.

С другой стороны, массив всегда гомогенный, т.е. он не может содержать элементы разных типов, что делает невозможным хранение там значений типов *int* и *double*.

Массив объектов *Object* создается по смещению 13, объект *Integer* добавляется в массив по смещению 22, объект *Double* добавляется в массив по смещению 29.

Предпоследняя инструкция *pop* удаляет элемент на **TOS!**, так что в момент исполнения *return*, стек пуст (или сбалансирован).

## Двухмерные массивы

Двухмерные массивы в Java это просто одномерные массивы *reference*-в на другие одномерные массивы.

Создадим двухмерный массив:

```
public static void main(String[] args)
{
    int[][] a = new int[5][10];
    a[1][2]=3;
}
```

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=3, locals=2, args_size=1
   0: iconst_5
   1: bipush    10
   3: multianewarray #2, 2    // class "[[I"
   7: astore_1
   8: aload_1
   9: iconst_1
  10: aaload
  11: iconst_2
  12: iconst_3
  13: iastore
  14: return
```

Он создается при помощи инструкции `multianewarray`: тип объекта и размерность передаются в операндах.

Размер массива (10\*5) остается в стеке (используя инструкции `iconst_5` и `bipush`).

*Reference* на строку #1 загружается по смещению 10 (`iconst_1` и `aaload`).

Выборка столбца происходит используя инструкцию `iconst_2` по смещению 11.

Значение для записи устанавливается по смещению 12.

`iastore` на 13 записывает элемент массива.

Как его прочитать?

```
public static int get12 (int[][] in)
{
    return in[1][2];
}
```

```
public static int get12(int[][]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=2, locals=1, args_size=1
   0: aload_0
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

1: iconst_1
2: aaload
3: iconst_2
4: iaload
5: ireturn

```

*Reference* на строку массива загружается по смещению 2, столбец устанавливается по смещению 3, *iaload* загружает элемент массива.

### Трехмерные массивы

Трехмерные массивы это просто одномерные массивы *reference*-ов на одномерные массивы *reference*-ов на одномерные массивы.

```

public static void main(String[] args)
{
    int[][][] a = new int[5][10][15];

    a[1][2][3]=4;

    get_elem(a);
}

```

```

public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=3, locals=2, args_size=1
   0: iconst_5
   1: bipush      10
   3: bipush      15
   5: multianewarray #2, 3    // class "[[[I"
   9: astore_1
  10: aload_1
  11: iconst_1
  12: aaload
  13: iconst_2
  14: aaload
  15: iconst_3
  16: iconst_4
  17: iastore
  18: aload_1
  19: invokestatic #3        // Method get_elem:([[[I)I
  22: pop
  23: return

```

Чтобы найти нужный *reference*, теперь нужно две инструкции *aaload*:

```

public static int get_elem (int[][][] a)
{
    return a[1][2][3];
}

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

public static int get_elem(int[][][]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=1, args_size=1
     0: aload_0
     1: iconst_1
     2: aaload
     3: iconst_2
     4: aaload
     5: iconst_3
     6: iaload
     7: ireturn

```

## Итоги

Возможно ли сделать переполнение буфера в Java?

Нет, потому что длина массива всегда присутствует в объекте массива, границы массива контролируются и при попытке выйти за границы, сработает исключение.

В Java нет многомерных массивов в том смысле, как в Си/Си++, так что Java не очень подходит для быстрых научных вычислений.

## 4.1.14. Строки

### Первый пример

Строки это объекты, и конструируются так же как и другие объекты (и массивы).

```

public static void main(String[] args)
{
    System.out.println("What is your name?");
    String input = System.console().readLine();
    System.out.println("Hello, "+input);
}

```

```

public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=2, args_size=1
     0: getstatic    #2          // Field java/lang/System.out:Ljava/io/
    ↪ PrintStream;
     3: ldc         #3          // String What is your name?
     5: invokevirtual #4          // Method java/io/PrintStream.println:(
    ↪ Ljava/lang/String;)V
     8: invokestatic #5          // Method java/lang/System.console:()
    ↪ Ljava/io/Console;
    11: invokevirtual #6          // Method java/io/Console.readLine:()
    ↪ Ljava/lang/String;

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

14: astore_1
15: getstatic    #2      // Field java/lang/System.out:Ljava/io/PrintStream;
↳ PrintStream;
18: new          #7      // class java/lang/StringBuilder
21: dup
22: invokespecial #8      // Method java/lang/StringBuilder.<init>:()V
↳ init>:()V
25: ldc          #9      // String Hello,
27: invokevirtual #10     // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
↳ append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
30: aload_1
31: invokevirtual #10     // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
↳ append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
34: invokevirtual #11     // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
↳ toString:()Ljava/lang/String;
37: invokevirtual #4      // Method java/io/PrintStream.println:(Ljava/lang/String;)V
↳ Ljava/lang/String;)V
40: return

```

Метод `readLine()` вызывается по смещению 11, *reference* на строку (введенную пользователем) остается на **TOS!**

По смещению 14, *reference* на строку сохраняется в первом слоте **LVA!**

Строка введенная пользователем перезагружается по смещению 30 и складывается со строкой «Hello, » используя класс `StringBuilder`.

Сконструированная строка затем выводится используя метод `println` по смещению 37.

## Второй пример

Еще один пример:

```

public class strings
{
    public static char test (String a)
    {
        return a.charAt(3);
    };

    public static String concat (String a, String b)
    {
        return a+b;
    }
}

```

```

public static char test(java.lang.String);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=1, args_size=1
 0: aload_0
 1: iconst_3

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
2: invokevirtual #2          // Method java/lang/String.charAt:(I)C
5: ireturn
```

Складывание строк происходит при помощи класса `StringBuilder`:

```
public static java.lang.String concat(java.lang.String, java.lang.String) ↵
↳ ;
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=2, args_size=2
      0: new          #3          // class java/lang/StringBuilder
      3: dup
      4: invokespecial #4          // Method java/lang/StringBuilder."<↳
↳ init>":()V
      7: aload_0
      8: invokevirtual #5          // Method java/lang/StringBuilder.↳
↳ append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
     11: aload_1
     12: invokevirtual #5          // Method java/lang/StringBuilder.↳
↳ append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
     15: invokevirtual #6          // Method java/lang/StringBuilder.↳
↳ toString:()Ljava/lang/String;
     18: areturn
```

Еще пример:

```
public static void main(String[] args)
{
    String s="Hello!";
    int n=123;
    System.out.println("s=" + s + " n=" + n);
}
```

И снова, строки создаются используя класс `StringBuilder` и его метод `append`, затем сконструированная строка передается в метод `println`:

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=3, args_size=1
      0: ldc          #2          // String Hello!
      2: astore_1
      3: bipush      123
      5: istore_2
      6: getstatic   #3          // Field java/lang/System.out:Ljava/io/↳
↳ PrintStream;
      9: new          #4          // class java/lang/StringBuilder
     12: dup
     13: invokespecial #5          // Method java/lang/StringBuilder."<↳
↳ init>":()V
     16: ldc          #6          // String s=
     18: invokevirtual #7          // Method java/lang/StringBuilder.↳
↳ append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

21: aload_1
22: invokevirtual #7      // Method java/lang/StringBuilder.↵
↳ append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
25: ldc          #8      // String n=
27: invokevirtual #7      // Method java/lang/StringBuilder.↵
↳ append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
30: iload_2
31: invokevirtual #9      // Method java/lang/StringBuilder.↵
↳ append:(I)Ljava/lang/StringBuilder;
34: invokevirtual #10     // Method java/lang/StringBuilder.↵
↳ toString:()Ljava/lang/String;
37: invokevirtual #11     // Method java/io/PrintStream.println:(↵
↳ Ljava/lang/String;)V
40: return

```

#### 4.1.15. Исключения

Немного переделаем пример *Month* (?? (стр. ??)):

Листинг 4.10: IncorrectMonthException.java

```

public class IncorrectMonthException extends Exception
{
    private int index;

    public IncorrectMonthException(int index)
    {
        this.index = index;
    }
    public int getIndex()
    {
        return index;
    }
}

```

Листинг 4.11: Month2.java

```

class Month2
{
    public static String[] months =
    {
        "January",
        "February",
        "March",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
        "December"
    }
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

};

public static String get_month (int i) throws ↵
↳ IncorrectMonthException
{
    if (i<0 || i>11)
        throw new IncorrectMonthException(i);
    return months[i];
};

public static void main (String[] args)
{
    try
    {
        System.out.println(get_month(100));
    }
    catch(IncorrectMonthException e)
    {
        System.out.println("incorrect month index: "+ e.↵
↳ getIndex());
        e.printStackTrace();
    }
};
}

```

Коротко говоря, `IncorrectMonthException.class` имеет только конструктор объекта и один метод-акцессор.

Класс `IncorrectMonthException` наследуется от `Exception`, так что конструктор `IncorrectMonthException` в начале вызывает конструктор класса `Exception`, затем он перекладывает входящее значение в единственное поле класса `IncorrectMonthException`:

```

public IncorrectMonthException(int);
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=2, args_size=2
     0: aload_0
     1: invokespecial #1          // Method java/lang/Exception."<init↵
↳ >":()V
     4: aload_0
     5: iload_1
     6: putfield      #2          // Field index:I
     9: return

```

`getIndex()` это просто акцессор.

*Reference* (указатель) на `IncorrectMonthException` передается в нулевом слоте **LVA!** (*this*), `aload_0` берет его, `getfield` загружает значение из объекта, `ireturn` возвращает его.

```

public int getIndex();
  flags: ACC_PUBLIC
  Code:

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

stack=1, locals=1, args_size=1
 0: aload_0
 1: getfield      #2          // Field index:I
 4: ireturn

```

Посмотрим на `get_month()` в `Month2.class`:

Листинг 4.12: `Month2.class`

```

public static java.lang.String get_month(int) throws ↵
↳ IncorrectMonthException;
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=3, locals=1, args_size=1
   0: iload_0
   1: iflt          10
   4: iload_0
   5: bipush       11
   7: if_icmple    19
  10: new          #2          // class IncorrectMonthException
  13: dup
  14: iload_0
  15: invokespecial #3          // Method IncorrectMonthException."<↳
↳ init>":(I)V
  18: athrow
  19: getstatic    #4          // Field months:[Ljava/lang/String;
  22: iload_0
  23: aaload
  24: areturn

```

`iflt` по смещению 1 это *if less than* (если меньше, чем).

В случае неправильного индекса, создается новый объект при помощи инструкции `new` по смещению 10.

Тип объекта передается как операнд инструкции (и это `IncorrectMonthException`).

Затем вызывается его конструктор, в который передается индекс (через **TOS!**) (по смещению 15).

В то время как управление находится на смещении 18, объект уже создан, теперь инструкция `athrow` берет указатель (*reference*) на только что созданный объект и сигнализирует в **JVM!**, чтобы тот нашел подходящий обработчик исключения.

Инструкция `athrow` не возвращает управление сюда, так что по смещению 19 здесь совсем другой **basic block**, не имеющий отношения к исключениям, сюда можно попасть со смещения 7.

Как работает обработчик? Посмотрим на `main()` в `Month2.class`:

Листинг 4.13: `Month2.class`

```

public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

Code:
  stack=3, locals=2, args_size=1
    0: getstatic    #5          // Field java/lang/System.out:Ljava/io/PrintStream;
    3: bipush       100
    5: invokestatic  #6          // Method get_month:(I)Ljava/lang/String;
    8: invokevirtual #7          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
   11: goto         47
   14: astore_1
   15: getstatic    #5          // Field java/lang/System.out:Ljava/io/PrintStream;
   18: new          #8          // class java/lang/StringBuilder
   21: dup
   22: invokespecial #9          // Method java/lang/StringBuilder.<init>:()V
   25: ldc          #10         // String incorrect month index:
   27: invokevirtual #11         // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
   30: aload_1
   31: invokevirtual #12         // Method IncorrectMonthException.getIndex:()I
   34: invokevirtual #13         // Method java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;
   37: invokevirtual #14         // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
   40: invokevirtual #7          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
   43: aload_1
   44: invokevirtual #15         // Method IncorrectMonthException.printStackTrace:()V
   47: return
Exception table:
   from   to   target type
    0     11   14   Class IncorrectMonthException

```

Тут есть Exception table, которая определяет, что между смещениями 0 и 11 (включительно) может случиться исключение IncorrectMonthException, и если это произойдет, то нужно передать управление на смещение 14.

Действительно, основная программа заканчивается на смещении 11.

По смещению 14 начинается обработчик, и сюда невозможно попасть, здесь нет никаких условных/безусловных переходов в эту область.

Но **JVM!** передаст сюда управление в случае исключения.

Самая первая astore\_1 (на 14) берет входящий указатель (*reference*) на объект исключения и сохраняет его в слоте 1 **LVA!**

Позже, по смещению 31 будет вызван метод этого объекта (getIndex()).

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



Указатель *reference* на текущий объект исключения передался немного раньше (смещение 30).

Остальной код это просто код для манипуляции со строками: в начале значение возвращенное методом `getIndex()` конвертируется в строку используя метод `toString()`, затем эта строка прибавляется к текстовой строке «incorrect month index: » (как мы уже рассматривали ранее), затем вызываются `println()` и `printStackTrace()`.

После того как `printStackTrace()` заканчивается, исключение уже обработано, мы можем возвращаться к нормальной работе.

По смещению 47 есть `return`, который заканчивает работу функции `main()`, но там может быть любой другой код, который исполнится, если исключения не произошло.

Вот пример, как IDA показывает интервалы исключений:

Листинг 4.14: из какого-то случайного найденного на компьютере автора `.class`-файла

```
.catch java/io/FileNotFoundException from met001_335 to met001_360\
using met001_360
  .catch java/io/FileNotFoundException from met001_185 to met001_214\
using met001_214
    .catch java/io/FileNotFoundException from met001_181 to met001_192\
using met001_195
      .catch java/io/FileNotFoundException from met001_155 to met001_176\
using met001_176
        .catch java/io/FileNotFoundException from met001_83 to met001_129 using ↵
          ↵ \
met001_129
            .catch java/io/FileNotFoundException from met001_42 to met001_66 using ↵
              ↵ \
met001_69
                .catch java/io/FileNotFoundException from met001_begin to met001_37\
using met001_37
```

#### 4.1.16. Классы

Простой класс:

Листинг 4.15: `test.java`

```
public class test
{
    public static int a;
    private static int b;

    public test()
    {
        a=0;
        b=0;
    }
}
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

public static void set_a (int input)
{
    a=input;
}
public static int get_a ()
{
    return a;
}
public static void set_b (int input)
{
    b=input;
}
public static int get_b ()
{
    return b;
}
}

```

Конструктор просто выставляет оба поля класса в нули:

```

public test();
  flags: ACC_PUBLIC
  Code:
    stack=1, locals=1, args_size=1
     0: aload_0
     1: invokespecial #1         // Method java/lang/Object."<init>":()V
    ↵ V
     4: iconst_0
     5: putstatic     #2         // Field a:I
     8: iconst_0
     9: putstatic     #3         // Field b:I
    12: return

```

Сеттер a:

```

public static void set_a(int);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
     0: iload_0
     1: putstatic     #2         // Field a:I
     4: return

```

Геттер a:

```

public static int get_a();
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=0, args_size=0
     0: getstatic     #2         // Field a:I
     3: ireturn

```

Сеттер b:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

public static void set_b(int);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
     0: iload_0
     1: putstatic    #3          // Field b:I
     4: return

```

Геттер b:

```

public static int get_b();
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=0, args_size=0
     0: getstatic    #3          // Field b:I
     3: ireturn

```

Здесь нет разницы между кодом, работающим для public-полей и private-полей. Но эта информация присутствует в .class-файле, и в любом случае невозможно иметь доступ к private-полям.

Создадим объект и вызовем метод:

Листинг 4.16: ex1.java

```

public class ex1
{
    public static void main(String[] args)
    {
        test obj=new test();
        obj.set_a (1234);
        System.out.println(obj.a);
    }
}

```

```

public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=2, args_size=1
     0: new          #2          // class test
     3: dup
     4: invokespecial #3          // Method test."<init>":()V
     7: astore_1
     8: aload_1
     9: pop
    10: sipush      1234
    13: invokestatic #4          // Method test.set_a:(I)V
    16: getstatic   #5          // Field java/lang/System.out:Ljava/io/PrintStream;
    19: aload_1
    20: pop
    21: getstatic   #6          // Field test.a:I

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

24: invokevirtual #7          // Method java/io/PrintStream.println:(L
↳ I)V
27: return

```

Инструкция `new` создает объект, но не вызывает конструктор (он вызывается по смещению 4).

Метод `set_a()` вызывается по смещению 16.

К полю `a` имеется доступ используя инструкцию `getstatic` по смещению 21.

#### 4.1.17. Простейшая модификация

##### Первый пример

Перейдем к простой задаче модификации кода.

```

public class nag
{
    public static void nag_screen()
    {
        System.out.println("This program is not registered");
    };
    public static void main(String[] args)
    {
        System.out.println("Greetings from the mega-software");
        nag_screen();
    }
}

```

Как можно избавиться от печати строки «This program is not registered»?

Наконец загрузим `.class`-файл в IDA:

```

; Segment type: Pure code
.method public static nag_screen()V
.limit stack 2
.line 4
178 000 002 |   getstatic java/lang/System.out Ljava/io/PrintStream; ; CODE XREF: main+8JP
018 003     |   ldc "This program is not registered"
182 000 004   invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
.line 5
177     |   return
??? ??? ???+ .end method
??? ??? ???+
???
; -----

; Segment type: Pure code
.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 1
.line 8
178 000 002   getstatic java/lang/System.out Ljava/io/PrintStream;
018 005     |   ldc "Greetings from the mega-software"
182 000 004   invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
.line 9
184 000 006   invokestatic nag.nag_screen()V
.line 10
177     |   return

```

Рис. 4.1: IDA

В начале заменим первый байт функции на 177 (это опкод инструкции return):

```

; Segment type: Pure code
.method public static nag_screen()V
.limit stack 2
.line 4

nag_screen: ; CODE XREF: main+8JP
177     |   return
000     |   0 ; 0x00
002     |   2 ; 0x02
018 003 |   ldc "This program is not registered"
182 000 004 |   invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
.line 5
177     |   return
??? ??? ???+ .end method
??? ??? ???+
???
; -----

```

Рис. 4.2: IDA

Но это не работает (JRE 1.7):

```

Exception in thread "main" java.lang.VerifyError: Expecting a stack map ↵
↳ frame
Exception Details:
Location:
  nag.nag_screen()V @1: nop
Reason:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

Error exists in the bytecode
Bytecode:
00000000: b100 0212 03b6 0004 b1

    at java.lang.Class.getDeclaredMethods0(Native Method)
    at java.lang.Class.privateGetDeclaredMethods(Class.java:2615)
    at java.lang.Class.getMethod0(Class.java:2856)
    at java.lang.Class.getMethod(Class.java:1668)
    at sun.launcher.LauncherHelper.getMainMethod(LauncherHelper.java↵
↵ :494)
    at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java↵
↵ :486)

```

Вероятно, в JVM есть проверки связанные с картами стека.

ОК, попробуем пропатчить её иначе, удаляя вызов функции `pag()`:

```

; Segment type: Pure code
.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 1
.line 8
178 000 002    getstatic java/lang/System.out Ljava/io/PrintStream;
018 005      ldc "Greetings from the mega-software"
182 000 004    invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
.line 9
000          nop
000          nop
000          nop
.line 10
177          return
; =====

```

Рис. 4.3: IDA

0 это опкод инструкции **NOP!**

Теперь всё работает!

## Второй пример

Еще один простой пример crackme:

```

public class password
{
    public static void main(String[] args)
    {
        System.out.println("Please enter the password");
        String input = System.console().readLine();
        if (input.equals("secret"))
            System.out.println("password is correct");
        else
            System.out.println("password is not correct");
    }
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
}
}
```

Загрузим в IDA:

```

; Segment type: Pure code
.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 2
.line 3
178 000 002   getstatic java/lang/System.out Ljava/io/PrintStream;
018 003     ldc "Please enter the password"
182 000 004   invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
.line 4
184 000 005   invokestatic java/lang/System.console()Ljava/io/Console;
182 000 006   invokevirtual java/io/Console.readLine()Ljava/lang/String;
076         astore_1 ; met002_slot001
.line 5
043         aload_1 ; met002_slot001
018 007     ldc "secret"
182 000 008   invokevirtual java/lang/String.equals(Ljava/lang/Object;)Z
153 000 014   ifeq met002_35
.line 6
178 000 002   getstatic java/lang/System.out Ljava/io/PrintStream;
018 009     ldc "password is correct"
182 000 004   invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
167 000 011   goto met002_43
.line 8
met002_35:                                     ; CODE XREF: main+21fj
178 000 002   .stack use locals
           locals Object java/lang/String
           .end stack
018 010     getstatic java/lang/System.out Ljava/io/PrintStream;
           ldc "password is not correct"
182 000 004   invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
           .line 9

```

Рис. 4.4: IDA

Видим здесь инструкцию `ifeq`, которая, собственно, всё и делает.

Её имя означает *if equal*, и это не очень удачное название, её следовало бы называть `ifz` (*if zero*), т.е. если значение на **TOS!** ноль, тогда совершить переход.

В нашем случае, переход происходит если пароль не верный (метод `equals` возвращает `False`, а это 0).

Первое что приходит в голову это пропатчить эту инструкцию.

В опкоде `ifeq` два байта, в которых закодировано смещение для перехода.

Чтобы инструкция не работала, мы должны установить байт 3 на третьем байте (потому что 3 будет прибавляться к текущему смещению, и в итоге переход будет на следующую инструкцию, ведь длина инструкции `ifeq` это 3 байта):

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; Segment type: Pure code
.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 2
.line 3
178 000 002   getstatic java/lang/System.out Ljava/io/PrintStream;
018 003     ldc "Please enter the password"
182 000 004   invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
.line 4
184 000 005   invokestatic java/lang/System.console()Ljava/io/Console;
182 000 006   invokevirtual java/io/Console.readLine()Ljava/lang/String;
076         astore_1 ; met002_slot001
.line 5
043         aload_1 ; met002_slot001
018 007     ldc "secret"
182 000 008   invokevirtual java/lang/String.equals(Ljava/lang/Object;)Z
153 000 003   ifeq met002_24
.line 6
met002_24:                                     ; CODE XREF: main+21↑j
178 000 002   getstatic java/lang/System.out Ljava/io/PrintStream;
018 009     ldc "password is correct"
182 000 004   invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
167 000 011   goto met002_43
.line 8
178 000 002   .stack use locals
                locals Object java/lang/String
                .end stack
                getstatic java/lang/System.out Ljava/io/PrintStream;
018 010     ldc "password is not correct"
182 000 004   invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
.line 9

```

Рис. 4.5: IDA

Это не работает (JRE 1.7):

```

Exception in thread "main" java.lang.VerifyError: Expecting a stackmap ↵
↳ frame at branch target 24
Exception Details:
Location:
    password.main([Ljava/lang/String;)V @21: ifeq
Reason:
    Expected stackmap frame at this location.
Bytecode:
    00000000: b200 0212 03b6 0004 b800 05b6 0006 4c2b
    00000010: 1207 b600 0899 0003 b200 0212 09b6 0004
    00000020: a700 0bb2 0002 120a b600 04b1
Stackmap Table:
    append_frame(@35,Object[#20])
    same_frame(@43)

    at java.lang.Class.getDeclaredMethods0(Native Method)
    at java.lang.Class.privateGetDeclaredMethods(Class.java:2615)
    at java.lang.Class.getMethod0(Class.java:2856)
    at java.lang.Class.getMethod(Class.java:1668)
    at sun.launcher.LauncherHelper.getMainMethod(LauncherHelper.java:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

↳ :494)
   at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java↵
↳ :486)

```

Хотя, надо сказать, работает в JRE 1.6.

Мы также можем попробовать заменить все три байта опкода `ifeq` на нулевые байты (**NOP!**), но это тоже не работает.

Видимо, начиная с JRE 1.7, там появилось больше проверок карт стека.

ОК, заменим весь вызов метода `equals` на инструкцию `iconst_1` плюс набор **NOP!**-ов:

```

; Segment type: Pure code
.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 2
.line 3
• 178 000 002   getstatic java/lang/System.out Ljava/io/PrintStream;
• 018 003     ldc "Please enter the password"
• 182 000 004   invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
.line 4
• 184 000 005   invokestatic java/lang/System.console()Ljava/io/Console;
• 182 000 006   invokevirtual java/io/Console.readLine()Ljava/lang/String;
• 076         astore_1 ; met002_slot001
.line 5
• 004         iconst_1
• 000         nop
• 000         nop
• 000         nop
• 000         nop
• 000         nop
--- 153 000 014   ifeq met002_35
.line 6
• 178 000 002   getstatic java/lang/System.out Ljava/io/PrintStream;
• 018 009     ldc "password is correct"
• 182 000 004   invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
• 167 000 011   goto met002_43
.line 8

met002_35:                                     ; CODE XREF: main+21↑j
178 000 002   .stack use locals
           locals Object java/lang/String
           .end stack

```

Рис. 4.6: IDA

1 будет всегда на **TOS!** во время исполнения инструкции `ifeq`, так что `ifeq` никогда не совершит переход.

Это работает.

#### 4.1.18. Итоги

Чего не хватает в Java в сравнении с Си/Си++?

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

- 
- Структуры: используйте классы.
  - Объединения (union): используйте иерархии классов.
  - Беззнаковые типы данных.

Кстати, из-за этого реализовывать криптографические алгоритмы на Java немного труднее.

- Указатели на функции.

## Глава 5

# Поиск в коде того что нужно

Современное ПО, в общем-то, минимализмом не отличается.

Но не потому, что программисты слишком много пишут, а потому что к исполняемым файлам обыкновенно прикомпилируют все подряд библиотеки. Если бы все вспомогательные библиотеки всегда выносили во внешние DLL, мир был бы иным. (Еще одна причина для Си++ — **STL!** и прочие библиотеки шаблонов.)

Таким образом, очень полезно сразу понимать, какая функция из стандартной библиотеки или более-менее известной (как Boost<sup>1</sup>, libpng<sup>2</sup>), а какая — имеет отношение к тому что мы пытаемся найти в коде.

Переписывать весь код на Си/Си++, чтобы разобраться в нем, безусловно, не имеет никакого смысла.

Одна из важных задач reverse engineer-а это быстрый поиск в коде того что собственно его интересует, а что - второстепенно.

Дизассемблер **IDA!** позволяет делать поиск как минимум строк, последовательностей байт, констант. Можно даже сделать экспорт кода в текстовый файл .lst или .asm и затем натравить на него grep, awk, и т. д.

Когда вы пытаетесь понять, что делает тот или иной код, это запросто может быть какая-то опен-сорсная библиотека вроде libpng. Поэтому, когда находите константы, или текстовые строки, которые выглядят явно знакомыми, всегда полезно их *погуглить*. А если вы найдете искомый опен-сорсный проект где это используется, то тогда будет достаточно будет просто сравнить вашу функцию с ней. Это решит часть проблем.

К примеру, если программа использует какие-то XML-файлы, первым шагом может быть установление, какая именно XML-библиотека для этого используется, ведь часто используется какая-то стандартная (или очень известная) вместо самодельной.

---

<sup>1</sup><http://www.boost.org/>

<sup>2</sup><http://www.libpng.org/pub/png/libpng.html>

К примеру, автор этих строк однажды пытался разобраться как происходит компрессия/декомпрессия сетевых пакетов в SAP 6.0. Это очень большая программа, но к ней идет подробный .PDB-файл с отладочной информацией, и это очень удобно. Он в конце концов пришел к тому что одна из функций декомпрессирующая пакеты называется CsDecomprLZC(). Не сильно раздумывая, он решил погуглить и оказалось, что функция с таким же названием имеется в MaxDB (это open-сорсный проект SAP) <sup>3</sup>.

<http://www.google.com/search?q=CsDecomprLZC>

Каково же было мое удивление, когда оказалось, что в MaxDB используется точно такой же алгоритм, скорее всего, с таким же исходником.

## 5.1. Идентификация исполняемых файлов

### 5.1.1. Microsoft Visual C++

Версии MSVC и DLL которые могут быть импортированы:

Маркетинговая вер.	Внутренняя вер.	Вер. CL.EXE	Импорт.DLL	Дата выхода
6	6.0	12.00	msvcrt.dll msvcp60.dll	June 1998
.NET (2002)	7.0	13.00	msvcr70.dll msvcp70.dll	February 13, 2002
.NET 2003	7.1	13.10	msvcr71.dll msvcp71.dll	April 24, 2003
2005	8.0	14.00	msvcr80.dll msvcp80.dll	November 7, 2005
2008	9.0	15.00	msvcr90.dll msvcp90.dll	November 19, 2007
2010	10.0	16.00	msvcr100.dll msvcp100.dll	April 12, 2010
2012	11.0	17.00	msvcr110.dll msvcp110.dll	September 12, 2012
2013	12.0	18.00	msvcr120.dll msvcp120.dll	October 17, 2013

msvcpr\*.dll содержит функции связанные с Си++, так что если она импортируется, скорее всего, вы имеете дело с программой на Си++.

### Name mangling

Имена обычно начинаются с символа ?.

О [name mangling](#) в MSVC читайте также здесь: [?? \(стр. ??\)](#).

<sup>3</sup>Больше об этом в соответствующей секции [?? \(стр. ??\)](#)

### 5.1.2. GCC

Кроме компиляторов под \*NIX, GCC имеется так же и для win32-окружения: в виде Cygwin и MinGW.

#### Name mangling

Имена обычно начинаются с символов `_Z`.

О [name mangling](#) в GCC читайте также здесь: ?? (стр. ??).

#### Cygwin

cygwin1.dll часто импортируется.

#### MinGW

msvcrt.dll может импортироваться.

### 5.1.3. Intel Fortran

libifcoremd.dll, libifportmd.dll и libiomp5md.dll (поддержка OpenMP) могут импортироваться.

В libifcoremd.dll много функций с префиксом `for_`, что значит *Fortran*.

### 5.1.4. Watcom, OpenWatcom

#### Name mangling

Имена обычно начинаются с символа `W`.

Например, так кодируется метод «method» класса «class» не имеющий аргументов и возвращающий `void`:

```
W?method$_class$_v
```

### 5.1.5. Borland

Вот пример [name mangling](#) в Borland Delphi и C++Builder:

```
@TApplication@IdleAction$qv
@TApplication@ProcessMDIAccels$qp6tagMSG
@TModule@$bctr$qpcpvt1
@TModule@$bdtr$qv
@TModule@ValidWindow$qp14TWindowsObject
@TrueColorTo8BitN$qpviiiiitliiiii
@TrueColorTo16BitN$qpviiiiitliiiii
@DIB24BitTo8BitBitmap$qpviiiiitliiiii
@TrueBitmap@$bctr$qpcl
@TrueBitmap@$bctr$qpvl
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
@TrueBitmap@bctr$qiilll
```

Имена всегда начинаются с символа @ затем следует имя класса, имя метода и закодированные типы аргументов.

Эти имена могут присутствовать с импортах .exe, экспортах .dll, отладочной информации, и т. д.

Borland Visual Component Libraries (VCL) находятся в файлах .bpl вместо .dll, например, vcl50.dll, rtl60.dll.

Другие DLL которые могут импортироваться: BORLNDMM.DLL.

## Delphi

Почти все исполняемые файлы имеют текстовую строку «Boolean» в самом начале сегмента кода, среди остальных имен типов.

Вот очень характерное для Delphi начало сегмента CODE, этот блок следует сразу за заголовком win32 PE-файла:

```

04 10 40 00 03 07 42 6f 6f 6c 65 61 6e 01 00 00 |..@...Boolean...|
00 00 01 00 00 00 00 10 40 00 05 46 61 6c 73 65 |.....@.False|
04 54 72 75 65 8d 40 00 2c 10 40 00 09 08 57 69 |.True.@.,.@...Wi|
64 65 43 68 61 72 03 00 00 00 00 ff ff 00 00 90 |deChar.....|
44 10 40 00 02 04 43 68 61 72 01 00 00 00 00 ff |D.@...Char.....|
00 00 00 90 58 10 40 00 01 08 53 6d 61 6c 6c 69 |....X.@...Smalli|
6e 74 02 00 80 ff ff ff 7f 00 00 90 70 10 40 00 |nt.....p.@.|
01 07 49 6e 74 65 67 65 72 04 00 00 00 80 ff ff |..Integer.....|
ff 7f 8b c0 88 10 40 00 01 04 42 79 74 65 01 00 |.....@...Byte..|
00 00 00 ff 00 00 00 90 9c 10 40 00 01 04 57 6f |.....@...Wo|
72 64 03 00 00 00 00 ff ff 00 00 90 b0 10 40 00 |rd.....@.|
01 08 43 61 72 64 69 6e 61 6c 05 00 00 00 ff |..Cardinal.....|
ff ff ff 90 c8 10 40 00 10 05 49 6e 74 36 34 00 |.....@...Int64.|
00 00 00 00 00 00 80 ff ff ff ff ff ff ff 7f 90 |.....|
e4 10 40 00 04 08 45 78 74 65 6e 64 65 64 02 90 |..@...Extended..|
f4 10 40 00 04 06 44 6f 75 62 6c 65 01 8d 40 00 |..@...Double..@.|
04 11 40 00 04 08 43 75 72 72 65 6e 63 79 04 90 |..@...Currency..|
14 11 40 00 0a 06 73 74 72 69 6e 67 20 11 40 00 |..@...string .@.|
0b 0a 57 69 64 65 53 74 72 69 6e 67 30 11 40 00 |..WideString0.@.|
0c 07 56 61 72 69 61 6e 74 8d 40 00 40 11 40 00 |..Variant.@.@.|
0c 0a 4f 6c 65 56 61 72 69 61 6e 74 98 11 40 00 |..OleVariant..@.|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
00 00 00 00 00 00 00 00 00 00 00 00 98 11 40 00 |.....@.|
04 00 00 00 00 00 00 00 18 4d 40 00 24 4d 40 00 |.....M@.$M@.|
28 4d 40 00 2c 4d 40 00 20 4d 40 00 68 4a 40 00 |(M@.,M@. M@.hJ@.|
84 4a 40 00 c0 4a 40 00 07 54 4f 62 6a 65 63 74 |.J@..J@..TObject|
a4 11 40 00 07 07 54 4f 62 6a 65 63 74 98 11 40 |..@...TObject..@|
00 00 00 00 00 00 06 53 79 73 74 65 6d 00 00 |.....System..|
c4 11 40 00 0f 0a 49 49 6e 74 65 72 66 61 63 65 |..@...IInterface|
00 00 00 00 01 00 00 00 00 00 00 00 c0 00 00 |.....|
00 00 00 00 46 06 53 79 73 74 65 6d 03 00 ff ff |....F.System....|
f4 11 40 00 0f 09 49 44 69 73 70 61 74 63 68 c0 |..@...IDispatch.|
11 40 00 01 00 04 02 00 00 00 00 c0 00 00 00 |.@.....|

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

00 00 00 46 06 53 79 73 74 65 6d 04 00 ff ff 90	...F.System.....
cc 83 44 24 04 f8 e9 51 6c 00 00 83 44 24 04 f8	..D\$...Ql...D\$..
e9 6f 6c 00 00 83 44 24 04 f8 e9 79 6c 00 00 cc	.ol...D\$...yl...
cc 21 12 40 00 2b 12 40 00 35 12 40 00 01 00 00	!.@.+.@.5.@....
00 00 00 00 00 00 00 00 00 c0 00 00 00 00 00 00	.....
46 41 12 40 00 08 00 00 00 00 00 00 00 8d 40 00	FA.@.....@.
bc 12 40 00 4d 12 40 00 00 00 00 00 00 00 00 00	..@.M.@.....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
bc 12 40 00 0c 00 00 00 4c 11 40 00 18 4d 40 00	..@.....L.@..M@.
50 7e 40 00 5c 7e 40 00 2c 4d 40 00 20 4d 40 00	P~@.\~@.,M@. M@.
6c 7e 40 00 84 4a 40 00 c0 4a 40 00 11 54 49 6e	l~@..J@..J@..TIn
74 65 72 66 61 63 65 64 4f 62 6a 65 63 74 8b c0	terfacedObject..
d4 12 40 00 07 11 54 49 6e 74 65 72 66 61 63 65	..@...TInterface
64 4f 62 6a 65 63 74 bc 12 40 00 a0 11 40 00 00	dObject...@...@..
00 06 53 79 73 74 65 6d 00 00 8b c0 00 13 40 00	..System.....@.
11 0b 54 42 6f 75 6e 64 41 72 72 61 79 04 00 00	..TBoundArray...
00 00 00 00 00 03 00 00 00 6c 10 40 00 06 53 79	.....l.@..Sy
73 74 65 6d 28 13 40 00 04 09 54 44 61 74 65 54	stem(.@...TDateT
69 6d 65 01 ff 25 48 e0 c4 00 8b c0 ff 25 44 e0	ime..%H.....%D.

Первые 4 байта сегмента данных (DATA) в исполняемых файлах могут быть 00 00 00 00, 32 13 8B C0 или FF FF FF FF. Эта информация может помочь при работе с запакованными/зашифрованными программами на Delphi.

### 5.1.6. Другие известные DLL

- vcomp\*.dll — Реализация OpenMP от Microsoft.

## 5.2. Связь с внешним миром (на уровне функции)

Очень желательно следить за аргументами ф-ции и возвращаемыми значениями, в отладчике или **DBI!**. Например, автор этих строк однажды пытался понять значение некоторой очень запутанной ф-ции, которая, как потом оказалось, была неверно реализованной пузырьковой сортировкой<sup>4</sup>. (Она работала правильно, но медленнее.) В то же время, наблюдение за входами и выходами этой ф-ции помогает мгновенно понять, что она делает.

Часто, когда вы видите деление через умножение (?? (стр. ??)), но забыли все детали о том, как оно работает, вы можете просто наблюдать за входом и выходом, и так быстро найти делитель.

## 5.3. Связь с внешним миром (win32)

Иногда, чтобы понять, что делает та или иная функция, можно её не разбирать, а просто посмотреть на её входы и выходы. Так можно сэкономить время.

<sup>4</sup>[https://yurichev.com/blog/weird\\_sort\\_KLEE/](https://yurichev.com/blog/weird_sort_KLEE/)

Обращения к файлам и реестру: для самого простого анализа может помочь утилита Process Monitor<sup>5</sup> от SysInternals.

Для анализа обращения программы к сети, может помочь Wireshark<sup>6</sup>.

Затем всё-таки придётся смотреть внутрь.

Первое на что нужно обратить внимание, это какие функции из **API! OS!** и какие функции стандартных библиотек используются. Если программа поделена на главный исполняемый файл и группу DLL-файлов, то имена функций в этих DLL, бывает так, могут помочь.

Если нас интересует, что именно приводит к вызову MessageBox() с определенным текстом, то первое, что можно попробовать сделать: найти в сегменте данных этот текст, найти ссылки на него, и найти, откуда может передаться управление к интересующему нас вызову MessageBox().

Если речь идет о компьютерной игре, и нам интересно какие события в ней более-менее случайны, мы можем найти функцию rand() или её заменитель (как алгоритм Mersenne twister), и посмотреть, из каких мест эта функция вызывается и что самое главное: как используется результат этой функции.

Один пример: ?? (стр. ??).

Но если это не игра, а rand() используется, то также весьма любопытно, зачем. Бывают неожиданные случаи вроде использования rand() в алгоритме для сжатия данных (для имитации шифрования): [blog.yurichev.com](http://blog.yurichev.com).

### 5.3.1. Часто используемые функции Windows API

Это функции которые можно увидеть в числе импортируемых. Но также нельзя забывать, что далеко не все они были использованы в коде написанном автором. Немалая часть может вызываться из библиотечных функций и **CRT!**-кода.

Многие ф-ции могут иметь суффикс -A для ASCII-версии и -W для Unicode-версии.

- Работа с реестром (advapi32.dll): RegEnumKeyEx, RegEnumValue, RegGetValue, RegOpenKeyEx, RegQueryValueEx.
- Работа с текстовыми .ini-файлами (kernel32.dll): GetPrivateProfileString.
- Диалоговые окна (user32.dll): MessageBox, MessageBoxEx, CreateDialog, SetDlgItemText, GetDlgItemText.
- Работа с ресурсами (?? (стр. ??)): (user32.dll): LoadMenu.
- Работа с TCP/IP-сетью (ws2\_32.dll): WSARcv, WSASend.
- Работа с файлами (kernel32.dll): CreateFile, ReadFile, ReadFileEx, WriteFile, WriteFileEx.
- Высокоуровневая работа с Internet (wininet.dll): WinHttpOpen.

<sup>5</sup><http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx>

<sup>6</sup><http://www.wireshark.org/>



- Проверка цифровой подписи исполняемого файла (wintrust.dll): WinVerifyTrust.
- Стандартная библиотека MSVC (в случае динамического связывания)(msvcr\*.dll): assert, itoa, ltoa, open, printf, read, strcmp, atol, atoi, fopen, fread, fwrite, memcmp, rand, strlen, strstr, strchr.

### 5.3.2. Расширение триального периода

Ф-ции доступа к реестру это частая цель тех, кто пытается расширить триальный период ПО, которое может сохранять дату/время инсталляции в реестре.

Другая популярная цель это ф-ции GetLocalTime() и GetSystemTime(): триальное ПО, при каждом запуске, должно как-то проверять текущую дату/время.

### 5.3.3. Удаление pag-окна

Популярный метод поиска того, что заставляет выводить pag-окно это перехват ф-ций MessageBox(), CreateDialog() и CreateWindow().

### 5.3.4. tracer: Перехват всех функций в отдельном модуле

В `tracer` есть поддержка точек останова INT3, хотя и срабатывающие только один раз, но зато их можно установить на все сразу функции в некоей DLL.

```
--one-time-INT3-bp:somedll.dll!.*
```

Либо, поставим INT3-прерывание на все функции, имена которых начинаются с префикса `xml`:

```
--one-time-INT3-bp:somedll.dll!xml.*
```

В качестве обратной стороны медали, такие прерывания срабатывают только один раз. Трасер покажет вызов какой-либо функции, если он случится, но только один раз. Еще один недостаток — увидеть аргументы функции также нельзя.

Тем не менее, эта возможность очень удобна для тех ситуаций, когда вы знаете что некая программа использует некую DLL, но не знаете какие именно функции в этой DLL.

И функций много.

Например, попробуем узнать, что использует `cygwin`-утилита `uptime`:

```
tracer -l:uptime.exe --one-time-INT3-bp:cygwin1.dll!.*
```

Так мы можем увидеть все функции из библиотеки `cygwin1.dll`, которые были вызваны хотя бы один раз, и откуда:

```
One-time INT3 breakpoint: cygwin1.dll!__main (called from uptime.exe!0EP+0↵  
↳ x6d (0x40106d))
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

One-time INT3 breakpoint: cygwin1.dll!_geteuid32 (called from uptime.exe!↵
↳ 0EP+0xba3 (0x401ba3))
One-time INT3 breakpoint: cygwin1.dll!_getuid32 (called from uptime.exe!0EP↵
↳ +0xbaa (0x401baa))
One-time INT3 breakpoint: cygwin1.dll!_getegid32 (called from uptime.exe!↵
↳ 0EP+0xcb7 (0x401cb7))
One-time INT3 breakpoint: cygwin1.dll!_getgid32 (called from uptime.exe!0EP↵
↳ +0xcbe (0x401cbe))
One-time INT3 breakpoint: cygwin1.dll!sysconf (called from uptime.exe!0EP+0↵
↳ x735 (0x401735))
One-time INT3 breakpoint: cygwin1.dll!setlocale (called from uptime.exe!0EP↵
↳ +0x7b2 (0x4017b2))
One-time INT3 breakpoint: cygwin1.dll!_open64 (called from uptime.exe!0EP+0↵
↳ x994 (0x401994))
One-time INT3 breakpoint: cygwin1.dll!_lseek64 (called from uptime.exe!0EP↵
↳ +0x7ea (0x4017ea))
One-time INT3 breakpoint: cygwin1.dll!read (called from uptime.exe!0EP+0↵
↳ x809 (0x401809))
One-time INT3 breakpoint: cygwin1.dll!sscanf (called from uptime.exe!0EP+0↵
↳ x839 (0x401839))
One-time INT3 breakpoint: cygwin1.dll!uname (called from uptime.exe!0EP+0↵
↳ x139 (0x401139))
One-time INT3 breakpoint: cygwin1.dll!time (called from uptime.exe!0EP+0↵
↳ x22e (0x40122e))
One-time INT3 breakpoint: cygwin1.dll!localtime (called from uptime.exe!0EP↵
↳ +0x236 (0x401236))
One-time INT3 breakpoint: cygwin1.dll!sprintf (called from uptime.exe!0EP+0↵
↳ x25a (0x40125a))
One-time INT3 breakpoint: cygwin1.dll!setutent (called from uptime.exe!0EP↵
↳ +0x3b1 (0x4013b1))
One-time INT3 breakpoint: cygwin1.dll!getutent (called from uptime.exe!0EP↵
↳ +0x3c5 (0x4013c5))
One-time INT3 breakpoint: cygwin1.dll!endutent (called from uptime.exe!0EP↵
↳ +0x3e6 (0x4013e6))
One-time INT3 breakpoint: cygwin1.dll!puts (called from uptime.exe!0EP+0↵
↳ x4c3 (0x4014c3))

```

## 5.4. Строки

### 5.4.1. Текстовые строки

#### Си/Си++

Обычные строки в Си заканчиваются нулем (**ASCIIZ!**-строки).

Причина, почему формат строки в Си именно такой (оканчивающийся нулем) вероятно историческая. В [Dennis M. Ritchie, *The Evolution of the Unix Time-sharing System*, (1979)] мы можем прочитать:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

A minor difference was that the unit of I/O was the word, not the byte, because the PDP-7 was a word-addressed machine. In practice this meant merely that all programs dealing with character streams ignored null characters, because null was used to pad a file to an even number of characters.

Строки выглядят в Hiew или FAR Manager точно так же:

```
int main()
{
    printf ("Hello, world!\n");
};
```

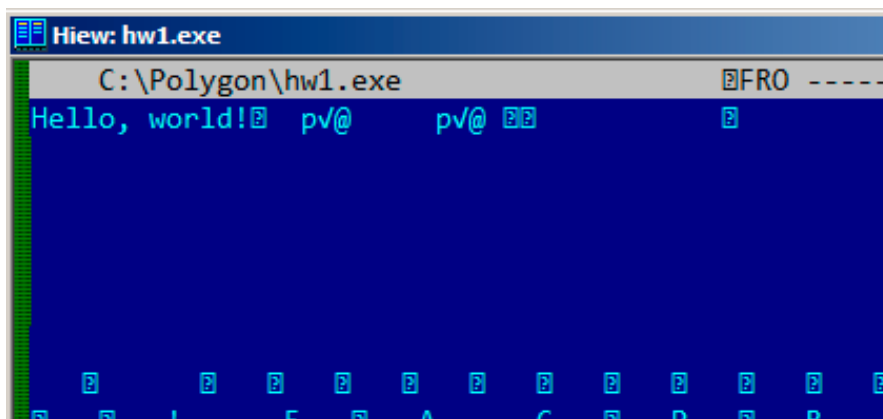


Рис. 5.1: Hiew

## Borland Delphi

Когда кодируются строки в Pascal и Delphi, сама строка предваряется 8-битным или 32-битным значением, в котором закодирована длина строки.

Например:

Листинг 5.1: Delphi

```
CODE:00518AC8          dd 19h
CODE:00518ACC aLoading__Plea db 'Loading... , please wait.',0
...
CODE:00518AFC          dd 10h
CODE:00518B00 aPreparingRun__ db 'Preparing run... ',0
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

## Unicode

Нередко уникодом называют все способы передачи символов, когда символ занимает 2 байта или 16 бит. Это распространенная терминологическая ошибка. Уникод — это стандарт, присваивающий номер каждому символу многих письменностей мира, но не описывающий способ кодирования.

Наиболее популярные способы кодирования: UTF-8 (наиболее часто используется в Интернете и \*NIX-системах) и UTF-16LE (используется в Windows).

## UTF-8

UTF-8 это один из очень удачных способов кодирования символов. Все символы латиницы кодируются так же, как и в ASCII-кодировке, а символы, выходящие за пределы ASCII-7-таблицы, кодируются несколькими байтами. 0 кодируется, как и прежде, нулевым байтом, так что все стандартные функции Си продолжают работать с UTF-8-строками так же как и с обычными строками.

Посмотрим, как символы из разных языков кодируются в UTF-8 и как это выглядит в FAR, в кодировке 437<sup>7</sup>:

How much? 100€?

```
(English) I can eat glass and it doesn't hurt me.
(Greek) Μπορώ να φάω σπασμένα γυαλιά χωρίς να πάθω τίποτα.
(Hungarian) Meg tudom enni az üveget, nem lesz tőle bajom.
(Icelandic) Ég get etið gler án þess að meiða mig.
(Polish) Mogę jeść szkło i mi nie szkodzi.
(Russian) Я могу есть стекло, оно мне не вредит.
(Arabic): أنا قادر على أكل الزجاج و هذا لا يؤلمني.
(Hebrew): אני יכול לאכול זכוכית וזה לא מזיק לי.
(Chinese) 我能吞下玻璃而不伤身体。
(Japanese) 私はガラスを食べられません。それは私を傷つけません。
(Hindi) मैं काँच खा सकता हूँ और मुझे उससे कोई चोट नहीं पहुंचती.
```

<sup>7</sup>Пример и переводы на разные языки были взяты здесь: <http://www.columbia.edu/~fdc/utf8/>

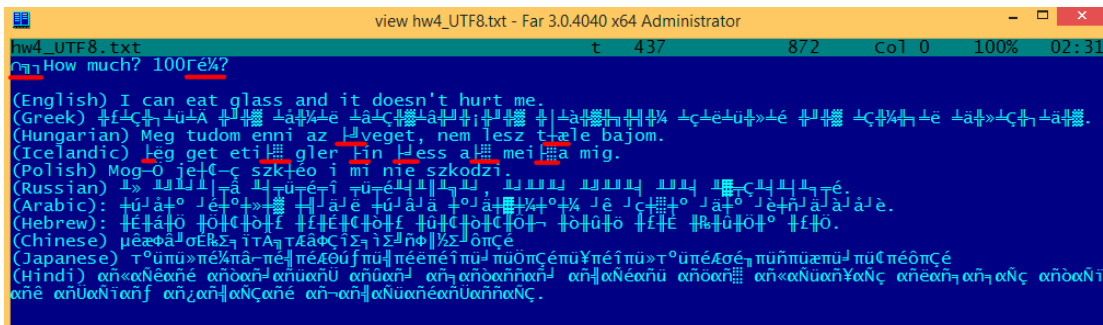


Рис. 5.2: FAR: UTF-8

Видно, что строка на английском языке выглядит точно так же, как и в ASCII-кодировке. В венгерском языке используются латиница плюс латинские буквы с диакритическими знаками. Здесь видно, что эти буквы кодируются несколькими байтами, они подчеркнуты красным. То же самое с исландским и польским языками. В самом начале имеется также символ валюты «Евро», который кодируется тремя байтами. Остальные системы письма здесь никак не связаны с латиницей. По крайней мере о русском, арабском, иврите и хинди мы можем сказать, что здесь видны повторяющиеся байты, что не удивительно, ведь, обычно буквы из одной и той же системы письменности расположены в одной или нескольких таблицах уникада, поэтому часто их коды начинаются с одних и тех же цифр.

В самом начале, перед строкой «How much?», видны три байта, которые на самом деле **ВОМ!**<sup>8</sup>. **ВОМ!** показывает, какой способ кодирования будет сейчас использоваться.

### UTF-16LE

Многие функции win32 в Windows имеют суффикс -A и -W. Первые функции работают с обычными строками, вторые с UTF-16LE-строками (*wide*). Во втором случае, каждый символ обычно хранится в 16-битной переменной типа *short*.

Строки с латинскими буквами выглядят в Niew или FAR как перемежающиеся с нулевыми байтами:

```
int wmain()
{
    wprintf (L"Hello, world!\n");
};
```

<sup>8</sup>ВОМ!

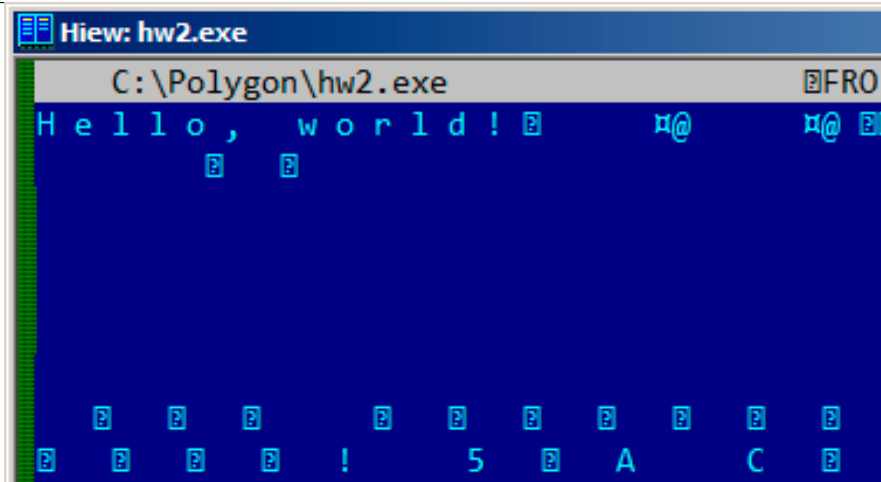


Рис. 5.3: Hiew

Подобное можно часто увидеть в системных файлах [Windows NT](#):

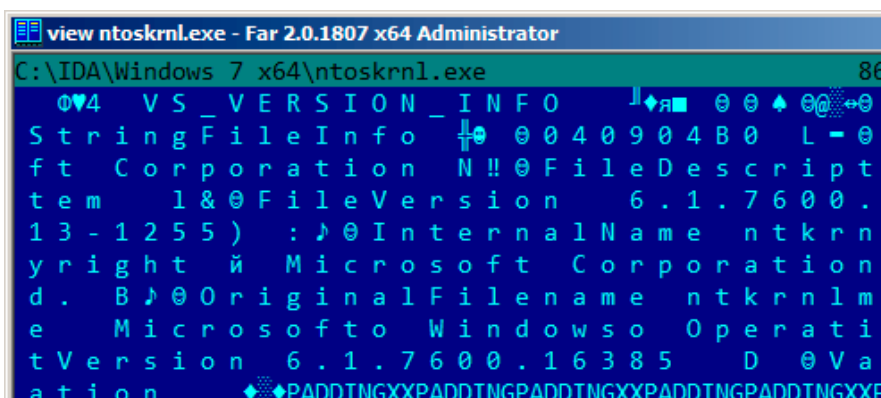


Рис. 5.4: Hiew

В **IDA!**, уником называется именно строки с символами, занимающими 2 байта:

```
.data:0040E000 aHelloWorld:
.data:0040E000          unicode 0, <Hello, world!>
.data:0040E000          dw 0Ah, 0
```

Вот как может выглядеть строка на русском языке («И снова здравствуйте!») закодированная в UTF-16LE:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

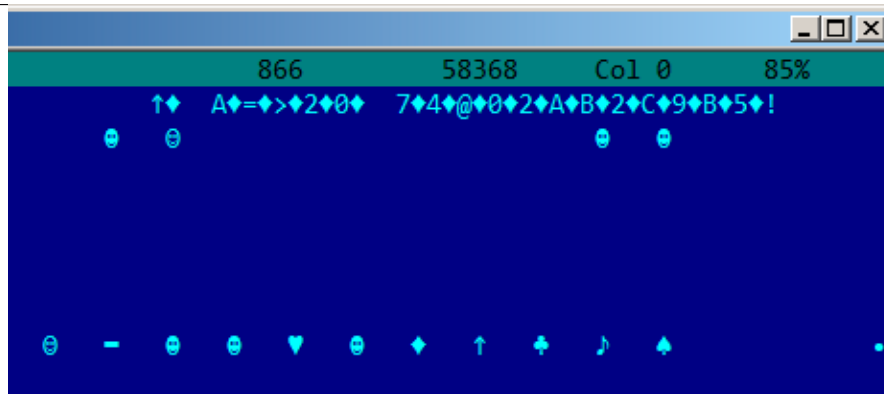


Рис. 5.5: Hiew: UTF-16LE

То что бросается в глаза — это то что символы перемежаются ромбиками (который имеет код 4). Действительно, в уникоде кирилличные символы находятся в четвертом блоке. Таким образом, все кирилличные символы в UTF-16LE находятся в диапазоне 0x400-0x4FF.

Вернемся к примеру, где одна и та же строка написана разными языками. Здесь посмотрим в кодировке UTF-16LE.

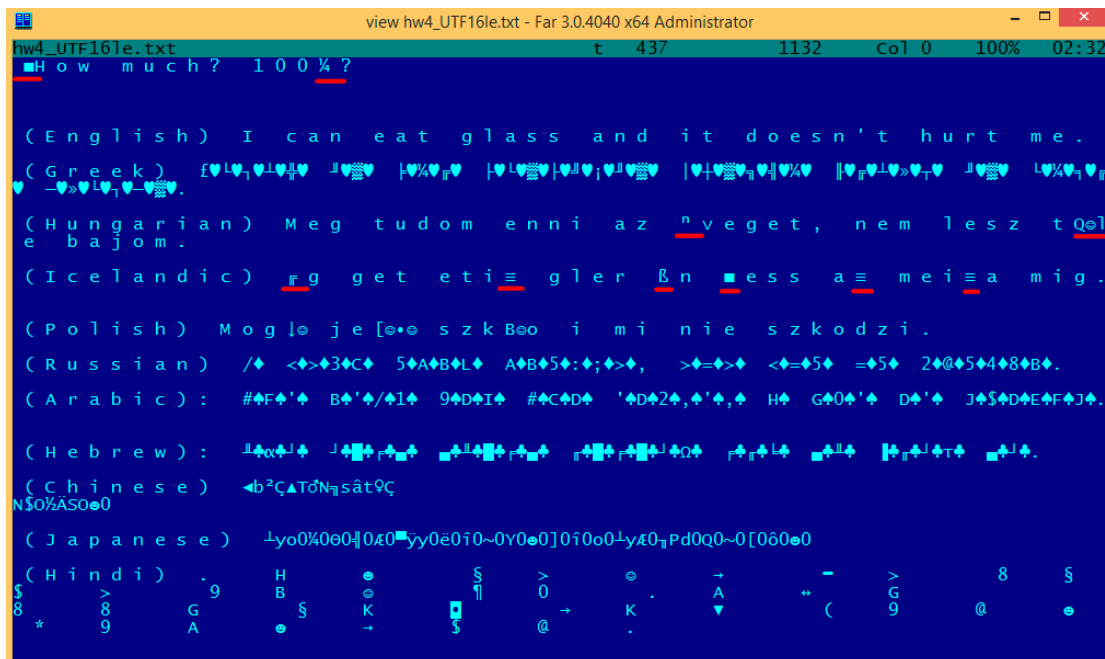


Рис. 5.6: FAR: UTF-16LE

Здесь мы также видим **ВОМ!** в самом начале. Все латинские буквы переме-

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

жаются с нулевыми байтами. Некоторые буквы с диакритическими знаками (венгерский и исландский языки) также подчеркнуты красным.

## Base64

Кодировка base64 очень популярна в тех случаях, когда нужно передать двоичные данные как текстовую строку.

По сути, этот алгоритм кодирует 3 двоичных байта в 4 печатаемых символа: все 26 букв латинского алфавита (в обоих регистрах), цифры, знак плюса («+») и слэша («/»), в итоге это получается 64 символа.

Одна отличительная особенность строк в формате base64, это то что они часто (хотя и не всегда) заканчиваются одним или двумя символами знака равенства («=») для выравнивания ([padding](#)), например:

```
AVjbbVSVfcUMu1xvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+qEJAp9lA0uWs=
```

```
WVjbbVSVfcUMu1xvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+qEJAp9lA0uQ==
```

Так что знак равенства («=») никогда не встречается в середине строк закодированных в base64.

Теперь пример кодирования вручную. Попробуем закодировать шестнадцатеричные байты 0x00, 0x11, 0x22, 0x33 в строку в формате base64:

```
$ echo -n "\x00\x11\x22\x33" | base64
ABEiMw==
```

Запишем все 4 байта в двоичной форме, затем перегруппируем их в 6-битные группы:

```
| 00 || 11 || 22 || 33 ||      ||      |
00000000000100010010001000110011??????????????
| A  || B  || E  || i  || M  || w  || =  || =  |
```

Первые три байта (0x00, 0x11, 0x22) можно закодировать в 4 base64-символа (“ABEi”), но последний (0x33) — нельзя, так что он кодируется используя два символа (“Mw”) и [padding](#)-символ (“=”) добавляется дважды, чтобы выровнять последнюю группу до 4-х символов. Таким образом, длина всех корректных base64-строк всегда делится на 4.

Base64 часто используется когда нужно закодировать двоичные данные в XML. PGP-ключи и подписи в “armored”-виде (т.е., в текстовом) кодируются в base64.

Некоторые люди пытаются использовать base64 для обфускации строк: <http://blog.sec-consult.com/2016/01/deliberately-hidden-backdoor-account-in.html><sup>9</sup>.

Существуют утилиты для сканирования бинарных файлов и нахождения в них base64-строк. Одна из них это base64scanner<sup>10</sup>.

<sup>9</sup><http://archive.is/nDCas>

<sup>10</sup><https://github.com/DennisYurichev/base64scanner>



Еще одна система кодирования, которая была более популярна в UseNet и FidoNet это Uuencoding. Двоичные файлы до сих пор кодируются при помощи Uuencode в журнале Phrack. Ее возможности почти такие же, но разница с base64 в том, что имя файла также передавалось в заголовке.

Кстати, у base64 есть близкий родственник - base32, алфавит которого состоит из 10 цифр и 26 латинских букв. Известное многим использование, это onion-адрес <sup>11</sup>, например:

<http://3g2upl4pq6kufc4m.onion/>. **URL!** не может содержать и строчные и заглавные латинские буквы, очевидно, по этой причине разработчики Tor использовали base32.

### 5.4.2. Поиск строк в бинарном файле

Actually, the best form of Unix documentation is frequently running the **strings** command over a program's object code. Using **strings**, you can get a complete list of the program's hard-coded file name, environment variables, undocumented options, obscure error messages, and so forth.

The Unix-Haters Handbook

Стандартная утилита в UNIX *strings* это самый простой способ увидеть строки в файле. Например, это строки найденные в исполняемом файле sshd из OpenSSH 7.2:

```
...
0123
0123456789
0123456789abcdefABCDEF.:/
%02x
...
%.100s, line %lu: Bad permitopen specification <%.100s>
%.100s, line %lu: invalid criteria
%.100s, line %lu: invalid tun device
...
%.200s/.ssh/environment
...
2886173b9c9b6fdbdeda7a247cd636db38deaa.debug
$2a$06$r3.juUaHZD\IbQa02dS9FuYxL1W9M81R1Tc92PoSNmzvpEqLkLGrK
...
3des-cbc
...
Bind to port %s on %s.
Bind to port %s on %s failed: %.200s.
/bin/login
/bin/sh
/bin/sh /etc/ssh/sshrC
```

<sup>11</sup><https://trac.torproject.org/projects/tor/wiki/doc/HiddenServiceNames>

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

...
D$4PQWR1
D$4PUj
D$4PV
D$4PVj
D$4PW
D$4PWj
D$4X
D$4XZj
D$4Y
...
diffie-hellman-group-exchange-sha1
diffie-hellman-group-exchange-sha256
digests
D$iPV
direct-streamlocal
direct-streamlocal@openssh.com
...
FFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A6...
...

```

Тут опции, сообщения об ошибках, пути к файлам, импортируемые модули, функции, и еще какие-то странные строки (ключи?) Присутствует также нечитаемый шум—иногда в x86-коде бывают целые куски состоящие из печатаемых ASCII-символов, вплоть до 8 символов.

Конечно, OpenSSH это open-сорсная программа. Но изучение читаемых строк внутри некоторого неизвестного бинарного файла это зачастую самый первый шаг в анализе.

Также можно использовать *grep*.

В Hiew есть такая же возможность (Alt-F6), также как и в Sysinternals ProcessMonitor.

### 5.4.3. Сообщения об ошибках и отладочные сообщения

Очень сильно помогают отладочные сообщения, если они имеются. В некотором смысле, отладочные сообщения, это отчет о том, что сейчас происходит в программе. Зачастую, это `printf()`-подобные функции, которые пишут куда-нибудь в лог, а бывает так что и не пишут ничего, но вызовы остались, так как эта сборка — не отладочная, а *release*.

Если в отладочных сообщениях дамвятся значения некоторых локальных или глобальных переменных, это тоже может помочь, как минимум, узнать их имена. Например, в Oracle RDBMS одна из таких функций: `ksdwrt()`.

Осмысленные текстовые строки вообще очень сильно могут помочь. Дизассемблер **IDA!** может сразу указать, из какой функции и из какого её места используется эта строка. Встречаются и смешные случаи <sup>12</sup>.

Сообщения об ошибках также могут помочь найти то что нужно. В Oracle RDBMS сигнализация об ошибках проходит при помощи вызова некоторой группы функ-

<sup>12</sup>[blog.yurichev.com](http://blog.yurichev.com)

ций.

Тут еще немного об этом: [blog.yurichev.com](http://blog.yurichev.com).

Можно довольно быстро найти, какие функции сообщают о каких ошибках, и при каких условиях.

Это, кстати, одна из причин, почему в защите софта от копирования, бывает так, что сообщение об ошибке заменяется невнятным кодом или номером ошибки. Мало кому приятно, если взломщик быстро поймет, из-за чего именно срабатывает защита от копирования, просто по сообщению об ошибке.

Один из примеров шифрования сообщений об ошибке, здесь: ?? (стр. ??).

#### 5.4.4. Подозрительные магические строки

Некоторые магические строки, используемые в бэкдорах выглядят очень подозрительно. Например, в домашних роутерах TP-Link WR740 был бэкдор<sup>13</sup>. Бэкдор активировался при посещении следующего URL: [http://192.168.0.1/userRpmNatDebugRpm26525557/start\\_art.html](http://192.168.0.1/userRpmNatDebugRpm26525557/start_art.html). Действительно, строка «userRpmNatDebugRpm26525557» присутствует в прошивке.

Эту строку нельзя было нагуглить до распространения информации о бэкдоре.

Вы не найдете ничего такого ни в одном **RFC!**<sup>14</sup>.

Вы не найдете ни одного алгоритма, который бы использовал такие странные последовательности байт.

И это не выглядит как сообщение об ошибке, или отладочное сообщение.

Так что проверить использование подобных странных строк — это всегда хорошая идея.

Иногда такие строки кодируются при помощи base64<sup>15</sup>. Так что неплохая идея их всех декодировать и затем просмотреть глазами, пусть даже бегло.

Более точно, такой метод сокрытия бэкдоров называется «security through obscurity» (безопасность через запутанность).

### 5.5. Вызовы `assert()`

Может также помочь наличие `assert()` в коде: обычно этот макрос оставляет название файла-источника, номер строки, и условие.

Наиболее полезная информация содержится в `assert`-условии, по нему можно судить по именам переменных или именам полей структур. Другая полезная

<sup>13</sup><http://sekurak.pl/tp-link-httpftp-backdoor/>, на русском: <http://m.habrahabr.ru/post/172799/>

<sup>14</sup>**RFC!**

<sup>15</sup>Например, бэкдор в кабельном модеме Arris: <http://www.securitylab.ru/analytics/461497.php>

информация — это имена файлов, по их именам можно попытаться предположить, что там за код. Также, по именам файлов можно опознать какую-либо очень известную open-сорсную библиотеку.

Листинг 5.2: Пример информативных вызовов assert()

```
.text:107D4B29 mov dx, [ecx+42h]
.text:107D4B2D cmp edx, 1
.text:107D4B30 jz short loc_107D4B4A
.text:107D4B32 push 1ECh
.text:107D4B37 push offset aWrite_c ; "write.c"
.text:107D4B3C push offset aTdTd_planarcon ;
    "td->td_planarconfig == PLANARCONFIG_CON"...
.text:107D4B41 call ds:_assert

...

.text:107D52CA mov edx, [ebp-4]
.text:107D52CD and edx, 3
.text:107D52D0 test edx, edx
.text:107D52D2 jz short loc_107D52E9
.text:107D52D4 push 58h
.text:107D52D6 push offset aDumpmode_c ; "dumpmode.c"
.text:107D52DB push offset aN30 ; "(n & 3) == 0"
.text:107D52E0 call ds:_assert

...

.text:107D6759 mov cx, [eax+6]
.text:107D675D cmp ecx, 0Ch
.text:107D6760 jle short loc_107D677A
.text:107D6762 push 2D8h
.text:107D6767 push offset aLzw_c ; "lzw.c"
.text:107D676C push offset aSpLzw_nbitsBit ; "sp->lzw_nbits <= BITS_MAX"
.text:107D6771 call ds:_assert
```

Полезно «гуглить» и условия и имена файлов, это может вывести вас к open-сорсной библиотеке. Например, если «погуглить» «sp->lzw\_nbits <= BITS\_MAX», это вполне предсказуемо выводит на open-сорсный код, что-то связанное с LZW-компрессией.

## 5.6. Константы

Люди, включая программистов, часто используют круглые числа вроде 10, 100, 1000, в т.ч. и в коде.

Практикующие реверсеры, обычно, хорошо знают их в шестнадцатеричном представлении: 10=0xA, 100=0x64, 1000=0x3E8, 10000=0x2710.

Иногда попадаются константы 0xA0000000 (0b10101010101010101010101010101010) и

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

0x55555555 (0b010101010101010101010101010101) — это чередующиеся биты. Это помогает отличить некоторый сигнал от сигнала где все биты включены (0b1111 ...) или выключены (0b0000 ...).

Например, константа 0x55AA используется как минимум в бут-секторе, **MBR!**<sup>16</sup>, и в **ROM!**<sup>17</sup> плат-расширений IBM-компьютеров.

Некоторые алгоритмы, особенно криптографические, используют хорошо различимые константы, которые при помощи **IDA!** легко находить в коде.

Например, алгоритм MD5 инициализирует свои внутренние переменные так:

```
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476
```

Если в коде найти использование этих четырех констант подряд — очень высокая вероятность что эта функция имеет отношение к MD5.

Еще такой пример это алгоритмы CRC16/CRC32, часто, алгоритмы вычисления контрольной суммы по CRC используют заранее заполненные таблицы, вроде:

Листинг 5.3: linux/lib/crc16.c

```
/** CRC table for the CRC-16. The poly is 0x8005 (x^16 + x^15 + x^2 + 1) */
u16 const crc16_table[256] = {
    0x0000, 0xC0C1, 0xC181, 0x0140, 0xC301, 0x03C0, 0x0280, 0xC241,
    0xC601, 0x06C0, 0x0780, 0xC741, 0x0500, 0xC5C1, 0xC481, 0x0440,
    0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81, 0x0E40,
    ...
}
```

См. также таблицу CRC32: ?? (стр. ??).

В бестабличных алгоритмах CRC используются хорошо известные полиномы, например 0xEDB88320 для CRC32.

### 5.6.1. Магические числа

Немало форматов файлов определяет стандартный заголовок файла где используются *магическое число* (magic number), один или даже несколько.

Скажем, все исполняемые файлы для Win32 и MS-DOS начинаются с двух символов «MZ».

В начале MIDI-файла должно быть «MThd». Если у нас есть использующая для чего-нибудь MIDI-файлы программа, наверняка она будет проверять MIDI-файлы на правильность хотя бы проверяя первые 4 байта.

Это можно сделать при помощи: (*buf* указывает на начало загруженного в память файла)

<sup>16</sup>**MBR!**

<sup>17</sup>**ROM!**

```
cmp [buf], 0x6468544D ; "MThd"
jnz _error_not_a_MIDI_file
```

...либо вызвав функцию сравнения блоков памяти `memcmp()` или любой аналогичный код, вплоть до инструкции `CMPSB` (?? (стр. ??)).

Найдя такое место мы получаем как минимум информацию о том, где начинается загрузка MIDI-файла, во-вторых, мы можем увидеть где располагается буфер с содержимым файла, и что еще оттуда берется, и как используется.

## Даты

Часто, можно встретить число вроде `0x19870116`, которое явно выглядит как дата (1987-й год, 1-й месяц (январь), 16-й день). Это может быть чей-то день рождения (программиста, его/ее родственника, ребенка), либо какая-то другая важная дата. Дата может быть записана и в другом порядке, например `0x16011987`. Даты в американском стиле также популярны, например `0x01161987`.

Известный пример это `0x19540119` (магическое число используемое в структуре суперблока UFS2), это день рождения Маршала Кирка МакКузика, видного разработчика FreeBSD.

В Stuxnet используется число "19790509" (хотя и не как 32-битное число, а как строка), и это привело к догадкам, что этот зловард связан с Израелем<sup>18</sup>.

Также, числа вроде таких очень популярны в любительской криптографии, например, это отрывок из внутренностей *секретной функции* донглы HASP3<sup>19</sup>:

```
void xor_pwd(void)
{
    int i;

    pwd ^= 0x09071966;
    for(i=0; i<8; i++)
    {
        al_buf[i] = pwd & 7; pwd = pwd >> 3;
    }
};

void emulate_func2(unsigned short seed)
{
    int i, j;
    for(i=0; i<8; i++)
    {
        ch[i] = 0;

        for(j=0; j<8; j++)
        {
            seed *= 0x1989;
            seed += 5;
        }
    }
}
```

<sup>18</sup>Это дата казни персидского еврея Habib Elghanian-a

<sup>19</sup><https://web.archive.org/web/20160311231616/http://www.woodmann.com/fravia/bayu3.htm>

```

        ch[i] |= (tab[(seed>>9)&0x3f]) << (7-j);
    }
}

```

## DHCP

Это касается также и сетевых протоколов. Например, сетевые пакеты протокола DHCP содержат так называемую *magic cookie*: 0x63538263. Какой-либо код, генерирующий пакеты по протоколу DHCP где-то и как-то должен внедрять в пакет также и эту константу. Найдя её в коде мы сможем найти место где происходит это и не только это. Любая программа, получающая DHCP-пакеты, должна где-то как-то проверять *magic cookie*, сравнивая это поле пакета с константой.

Например, берем файл `dhcpcore.dll` из Windows 7 x64 и ищем эту константу. И находим, два раза: оказывается, эта константа используется в функциях с красноречивыми названиями

`DhcpExtractOptionsForValidation()` и `DhcpExtractFullOptions()`:

Листинг 5.4: `dhcpcore.dll` (Windows 7 x64)

```

.rdata:000007FF6483CBE8 dword_7FF6483CBE8 dd 63538263h ; DATA XREF:
DhcpExtractOptionsForValidation+79
.rdata:000007FF6483CBEC dword_7FF6483CBEC dd 63538263h ; DATA XREF:
DhcpExtractFullOptions+97

```

А вот те места в функциях где происходит обращение к константам:

Листинг 5.5: `dhcpcore.dll` (Windows 7 x64)

```

.text:000007FF6480875F mov     eax, [rsi]
.text:000007FF64808761 cmp     eax, cs:dword_7FF6483CBE8
.text:000007FF64808767 jnz     loc_7FF64817179

```

И:

Листинг 5.6: `dhcpcore.dll` (Windows 7 x64)

```

.text:000007FF648082C7 mov     eax, [r12]
.text:000007FF648082CB cmp     eax, cs:dword_7FF6483CBEC
.text:000007FF648082D1 jnz     loc_7FF648173AF

```

### 5.6.2. Специфические константы

Иногда, бывают какие-то специфические константы для некоторого типа кода. Например, однажды автор сих строк пытался разобраться с кодом, где подзрительно часто встречалось число 12. Размеры многих массивов также были 12, или кратные 12 (24, и т. д.). Оказалось, этот код брал на вход 12-канальный аудиофайл и обрабатывал его.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

И наоборот: например, если программа работает с текстовым полем длиной 120 байт, значит где-то в коде должна быть константа 120, или 119. Если используется UTF-16, то тогда  $2 \cdot 120$ . Если код работает с сетевыми пакетами фиксированной длины, то хорошо бы и такую константу поискать в коде.

Это также справедливо для любительской криптографии (ключи с лицензией, и т. д.). Если зашифрованный блок имеет размер в  $n$  байт, вы можете попробовать поискать это число в коде. Также, если вы видите фрагмент кода, который при исполнении, повторяется  $n$  раз в цикле, это может быть функция шифрования/дешифрования.

### 5.6.3. Поиск констант

В **IDA!** это очень просто, Alt-B или Alt-I.

А для поиска константы в большом количестве файлов, либо для поиска их в неисполняемых файлах, имеется небольшая утилита *binary grep*<sup>20</sup>.

## 5.7. Поиск нужных инструкций

Если программа использует инструкции сопроцессора, и их не очень много, то можно попробовать вручную проверить отладчиком какую-то из них.

К примеру, нас может заинтересовать, при помощи чего Microsoft Excel считает результаты формул, введенных пользователем. Например, операция деления.

Если загрузить excel.exe (из Office 2010) версии 14.0.4756.1000 в **IDA!**, затем сделать полный листинг и найти все инструкции FDIV (но кроме тех, которые в качестве второго операнда используют константы — они, очевидно, не подходят нам):

```
cat EXCEL.lst | grep fdiv | grep -v dbl_ > EXCEL.fdiv
```

...то окажется, что их всего 144.

Мы можем вводить в Excel строку вроде  $= (1/3)$  и проверить все эти инструкции.

Проверяя каждую инструкцию в отладнике или *tracer* (проверять эти инструкции можно по 4 за раз), окажется, что нам везет и срабатывает всего лишь 14-я по счету:

```
.text:3011E919 DC 33          fdiv    qword ptr [ebx]
```

```
PID=13944|TID=28744|(0) 0x2f64e919 (Excel.exe!BASE+0x11e919)
EAX=0x02088006 EBX=0x02088018 ECX=0x00000001 EDX=0x00000001
ESI=0x02088000 EDI=0x00544804 EBP=0x0274FA3C ESP=0x0274F9F8
EIP=0x2F64E919
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
```

<sup>20</sup>[GitHub](#)



```
FPU StatusWord=
FPU ST(0): 1.000000
```

В ST(0) содержится первый аргумент (1), второй содержится в [EBX].

Следующая за FDIV инструкция (FSTP) записывает результат в память:

```
.text:3011E91B DD 1E          fstp    qword ptr [esi]
```

Если поставить breakpoint на ней, то мы можем видеть результат:

```
PID=32852|TID=36488|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00598006 EBX=0x00598018 ECX=0x00000001 EDX=0x00000001
ESI=0x00598000 EDI=0x00294804 EBP=0x026CF93C ESP=0x026CF8F8
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
```

А также, в рамках пранка<sup>21</sup>, модифицировать его на лету:

```
tracer -l:excel.exe bpx=excel.exe!BASE+0x11E91B,set(st0,666)
```

```
PID=36540|TID=24056|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00680006 EBX=0x00680018 ECX=0x00000001 EDX=0x00000001
ESI=0x00680000 EDI=0x00395404 EBP=0x0290FD9C ESP=0x0290FD58
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
Set ST0 register to 666.000000
```

Excel показывает в этой ячейке 666, что окончательно убеждает нас в том, что мы нашли нужное место.

<sup>21</sup>practical joke

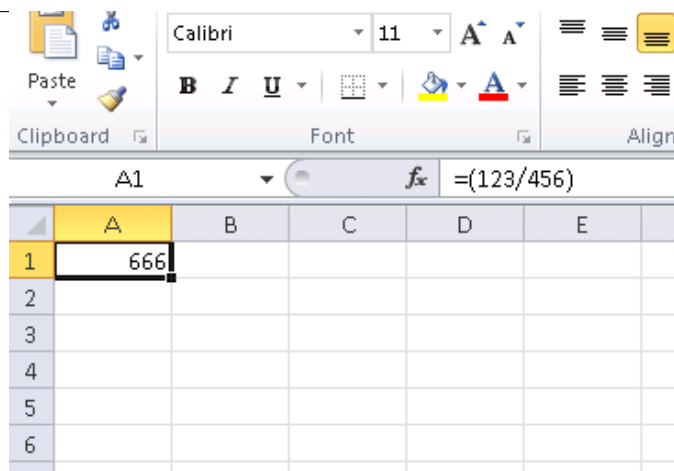


Рис. 5.7: Пранк сработал

Если попробовать ту же версию Excel, только x64, то окажется что там инструкций FDIV всего 12, причем нужная нам — третья по счету.

```
tracer.exe -l:excel.exe bpx=excel.exe!BASE+0x1B7FCC,set(st0,666)
```

Видимо, все дело в том, что много операций деления переменных типов *float* и *double* компилятор заменил на SSE-инструкции вроде DIVSD, коих здесь теперь действительно много (DIVSD присутствует в количестве 268 инструкций).

## 5.8. Подозрительные паттерны кода

### 5.8.1. Инструкции XOR

Инструкции вроде XOR op, op (например, XOR EAX, EAX) обычно используются для обнуления регистра, однако, если операнды разные, то применяется операция именно «исключающего или». Эта операция очень редко применяется в обычном программировании, но применяется очень часто в криптографии, включая любительскую.

Особенно подозрительно, если второй операнд — это большое число. Это может указывать на шифрование, вычисление контрольной суммы, и т. д.

Одно из исключений из этого наблюдения о котором стоит сказать, то, что генерация и проверка значения «канарейки» (?? (стр. ??)) часто происходит, используя инструкцию XOR.

Этот AWK-скрипт можно использовать для обработки листингов (.lst) созданных **IDA!**:

```
gawk -e '$2=="xor" { tmp=substr($3, 0, length($3)-1); if (tmp!=$4) if ($4!="\n"
↳ esp") if ($4!="ebp") { print $1, $2, tmp, ",", $4 } }' filename.lst
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Нельзя также забывать, что подобный скрипт может захватить и неверно дизассемблированный код (?? (стр. ??)).

### 5.8.2. Вручную написанный код на ассемблере

Современные компиляторы не генерируют инструкции LOOP и RCL. С другой стороны, эти инструкции хорошо знакомы кодерам, предпочитающим писать прямо на ассемблере. Подобные инструкции отмечены как (M) в списке инструкций в приложении: ?? (стр. ??). Если такие инструкции встретились, можно сказать с какой-то вероятностью, что этот фрагмент кода написан вручную.

Также, пролог/эпилог функции обычно не встречается в ассемблерном коде, написанном вручную.

Как правило, во вручную написанном коде, нет никакого четкого метода передачи аргументов в функцию.

Пример из ядра Windows 2003 (файл ntoskrnl.exe):

```

MultiplyTest proc near                ; CODE XREF: Get386Stepping
xor     cx, cx
loc_620555:                          ; CODE XREF: MultiplyTest+E
    push    cx
    call   Multiply
    pop     cx
    jb     short locret_620563
    loop   loc_620555
    cld
locret_620563:                       ; CODE XREF: MultiplyTest+C
    retn
MultiplyTest endp

Multiply    proc near                ; CODE XREF: MultiplyTest+5
    mov     ecx, 81h
    mov     eax, 417A000h
    mul    ecx
    cmp     edx, 2
    stc
    jnz    short locret_62057F
    cmp     eax, 0FE7A000h
    stc
    jnz    short locret_62057F
    cld
locret_62057F:                       ; CODE XREF: Multiply+10
                                         ; Multiply+18
    retn
Multiply    endp

```

Действительно, если заглянуть в исходные коды **WRK!**<sup>22</sup> v1.2, данный код можно найти в файле *WRK-v1.2\base\ntos\ke\i386\cpu.asm*.

<sup>22</sup>**WRK!**

А инструкцию RCL, я смог найти в файле ntoskrnl.exe из Windows 2003 x86 (компилятор MS Visual C). Она встречается только один раз, в ф-ции RtlExtendedLargeIntegerDivide(), и это может быть вставка на ассемблере.

## 5.9. Использование magic numbers для трассировки

Нередко бывает нужно узнать, как используется то или иное значение, прочитанное из файла либо взятое из пакета, принятого по сети. Часто, ручное слежение за нужной переменной это трудный процесс. Один из простых методов (хотя и не полностью надежный на 100%) это использование вашей собственной *magic number*.

Это чем-то напоминает компьютерную томографию: пациенту перед сканированием вводят в кровь рентгеноконтрастный препарат, хорошо отсвечивающий в рентгеновских лучах. Известно, как кровь нормального человека расходится, например, по почкам, и если в этой крови будет препарат, то при томографии будет хорошо видно, достаточно ли хорошо кровь расходится по почкам и нет ли там камней, например, и прочих образований.

Мы можем взять 32-битное число вроде 0x0badf00d, либо чью-то дату рождения вроде 0x11101979 и записать это, занимающее 4 байта число, в какое-либо место файла используемого исследуемой нами программой.

Затем, при трассировке этой программы, в том числе, при помощи [tracer](#) в режиме *code coverage*, а затем при помощи *grep* или простого поиска по текстовому файлу с результатами трассировки, мы можем легко увидеть, в каких местах кода использовалось это значение, и как.

Пример результата работы [tracer](#) в режиме *сс*, к которому легко применить утилиту *grep*:

```
0x150bf66 (_kziaia+0x14), e=      1 [MOV EBX, [EBP+8]] [EBP+8]=0xf59c934
0x150bf69 (_kziaia+0x17), e=      1 [MOV EDX, [69AEB08h]] [69AEB08h]=0
0x150bf6f (_kziaia+0x1d), e=      1 [FS: MOV EAX, [2Ch]]
0x150bf75 (_kziaia+0x23), e=      1 [MOV ECX, [EAX+EDX*4]] [EAX+EDX*4]=0x
  ↪ xf1ac360
0x150bf78 (_kziaia+0x26), e=      1 [MOV [EBP-4], ECX] ECX=0xf1ac360
```

Это справедливо также и для сетевых пакетов. Важно только, чтобы наш *magic number* был как можно более уникален и не присутствовал в самом коде.

Помимо [tracer](#), такой эмулятор MS-DOS как DosBox, в режиме *heavydebug*, может писать в отчет информацию обо всех состояниях регистра на каждом шаге исполнения программы<sup>23</sup>, так что этот метод может пригодиться и для исследования программ под DOS.

<sup>23</sup>См. также мой пост в блоге об этой возможности в DosBox: [blog.yurichev.com](http://blog.yurichev.com)

## 5.10. Циклы

Когда ваша программа работает с некоторым файлом, или буфером некоторой длины, внутри кода где-то должен быть цикл с дешифровкой/обработкой.

Вот реальный пример выхода инструмента [tracer](#). Был код, загружающий некоторый зашифрованный файл размером 258 байт. Я могу запустить его с целью подсчета, сколько раз была исполнена каждая инструкция (в наше время **DBI!** послужила бы куда лучше). И я могу быстро найти место в коде, которое было исполнено 259/258 раз:

```

...
0x45a6b5 e= 1 [FS: MOV [0], EAX] EAX=0x218fb08
0x45a6bb e= 1 [MOV [EBP-254h], ECX] ECX=0x218fbd8
0x45a6c1 e= 1 [MOV EAX, [EBP-254h]] [EBP-254h]=0x218fbd8
0x45a6c7 e= 1 [CMP [EAX+14h], 0] [EAX+14h]=0x102
0x45a6cb e= 1 [JZ 45A9F2h] ZF=false
0x45a6d1 e= 1 [MOV [EBP-0Dh], 1]
0x45a6d5 e= 1 [XOR ECX, ECX] ECX=0x218fbd8
0x45a6d7 e= 1 [MOV [EBP-14h], CX] CX=0
0x45a6db e= 1 [MOV [EBP-18h], 0]
0x45a6e2 e= 1 [JMP 45A6EDh]
0x45a6e4 e= 258 [MOV EDX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0x
  ↪ xfd..0x101
0x45a6e7 e= 258 [ADD EDX, 1] EDX=0..5 (248 items skipped) 0xfd..0x101
0x45a6ea e= 258 [MOV [EBP-18h], EDX] EDX=1..6 (248 items skipped) 0xfe..0x
  ↪ x102
0x45a6ed e= 259 [MOV EAX, [EBP-254h]] [EBP-254h]=0x218fbd8
0x45a6f3 e= 259 [MOV ECX, [EBP-18h]] [EBP-18h]=0..5 (249 items skipped) 0x
  ↪ xfe..0x102
0x45a6f6 e= 259 [CMP ECX, [EAX+14h]] ECX=0..5 (249 items skipped) 0xfe..0x
  ↪ x102 [EAX+14h]=0x102
0x45a6f9 e= 259 [JNB 45A727h] CF=false,true
0x45a701 e= 259 [MOV EDX, [EBP-254h]] [EBP-254h]=0x218fbd8
0x45a701 e= 258 [MOV EAX, [EDX+10h]] [EDX+10h]=0x21ee4c8
0x45a704 e= 258 [MOV ECX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0x
  ↪ xfd..0x101
0x45a707 e= 258 [ADD ECX, 1] ECX=0..5 (248 items skipped) 0xfd..0x101
0x45a70a e= 258 [IMUL ECX, ECX, 1Fh] ECX=1..6 (248 items skipped) 0xfe..0x
  ↪ x102
0x45a70d e= 258 [MOV EDX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0x
  ↪ xfd..0x101
0x45a710 e= 258 [MOVZX EAX, [EAX+EDX]] [EAX+EDX]=1..6 (156 items skipped) 0x
  ↪ xf3, 0xf8, 0xf9, 0xfc, 0xfd
0x45a714 e= 258 [XOR EAX, ECX] EAX=1..6 (156 items skipped) 0xf3, 0xf8, 0x
  ↪ xf9, 0xfc, 0xfd ECX=0x1f, 0x3e, 0x5d, 0x7c, 0x9b (248 items skipped) ↪
  ↪ 0x1ec2, 0x1ee1, 0x1f00, 0x1f1f, 0x1f3e
0x45a716 e= 258 [MOV ECX, [EBP-254h]] [EBP-254h]=0x218fbd8
0x45a71c e= 258 [MOV EDX, [ECX+10h]] [ECX+10h]=0x21ee4c8
0x45a71f e= 258 [MOV ECX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0x
  ↪ xfd..0x101
0x45a722 e= 258 [MOV [EDX+ECX], AL] AL=0..5 (77 items skipped) 0xe2, 0xee, ↪
  ↪ 0xef, 0xf7, 0xfc

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
0x45a725 e= 258 [JMP 45A6E4h]
0x45a727 e= 1 [PUSH 5]
0x45a729 e= 1 [MOV ECX, [EBP-254h]] [EBP-254h]=0x218fbd8
0x45a72f e= 1 [CALL 45B500h]
0x45a734 e= 1 [MOV ECX, EAX] EAX=0x218fbd8
0x45a736 e= 1 [CALL 45B710h]
0x45a73b e= 1 [CMP EAX, 5] EAX=5
...

```

Как потом оказалось, это цикл дешифрования.

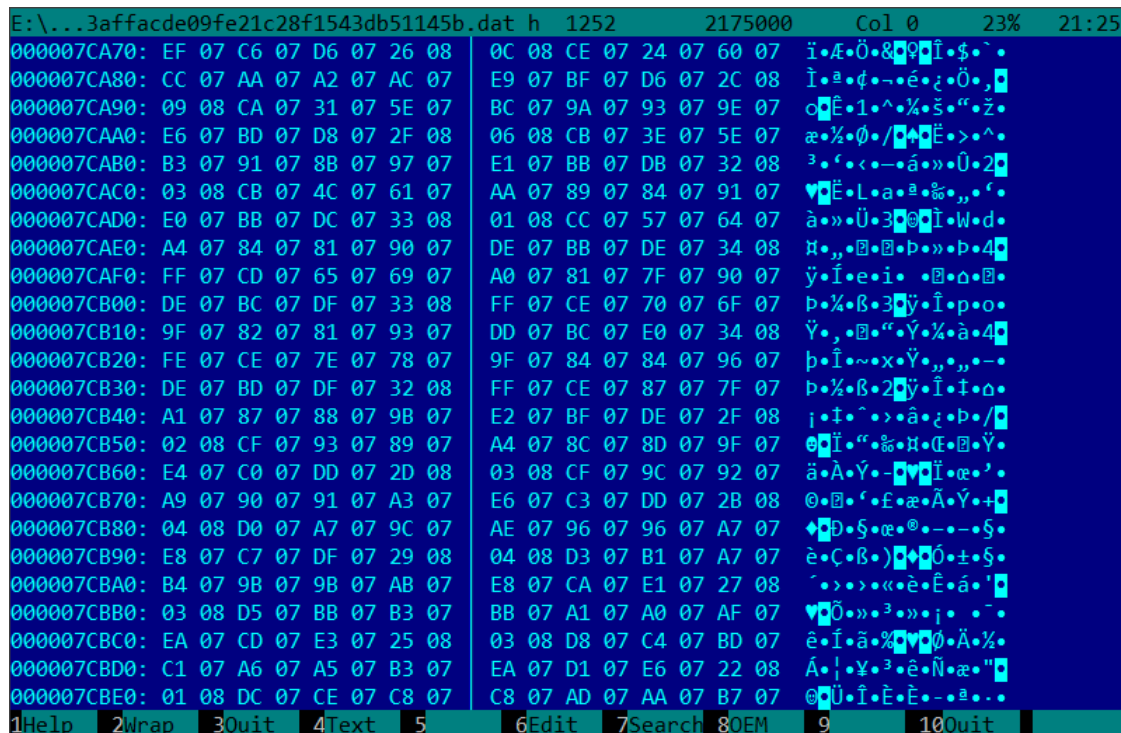
### 5.10.1. Некоторые паттерны в бинарных файлах

Все примеры здесь были подготовлены в Windows с активной кодовой страницей 437 в консоли. Двоичные файлы внутри могут визуально выглядеть иначе если установлена другая кодовая страница.

## Массивы

Иногда мы можем легко заметить массив 16/32/64-битных значений визуально, в шестнадцатеричном редакторе.

Вот пример массива 16-битных значений. Мы видим, что каждый первый байт в паре всегда равен 7 или 8, а второй выглядит случайным:



```
E:\...\3affacde09fe21c28f1543db51145b.dat h 1252 2175000 Col 0 23% 21:25
000007CA70: EF 07 C6 07 D6 07 26 08 0C 08 CE 07 24 07 60 07 i•Æ•Ö•&•&•Ï•$•`•
000007CA80: CC 07 AA 07 A2 07 AC 07 E9 07 BF 07 D6 07 2C 08 Ì•à•φ•→•é•¿•Ö•,•
000007CA90: 09 08 CA 07 31 07 5E 07 BC 07 9A 07 93 07 9E 07 o•Ë•1•^•%•§•“•Ž•
000007CAA0: E6 07 BD 07 D8 07 2F 08 06 08 CB 07 3E 07 5E 07 æ•%•Ø•/••Ë•>•^•
000007CAB0: B3 07 91 07 8B 07 97 07 E1 07 BB 07 DB 07 32 08 ³•‘•<•-•á•»•Û•2•
000007CAC0: 03 08 CB 07 4C 07 61 07 AA 07 89 07 84 07 91 07 ♥•Ë•L•a•à•••,••‘•
000007CAD0: E0 07 BB 07 DC 07 33 08 01 08 CC 07 57 07 64 07 à•»•Û•3••Ë•Ï•W•d•
000007CAE0: A4 07 84 07 81 07 90 07 DE 07 BB 07 DE 07 34 08 ð•„•••••b•»•p•4•
000007CAF0: FF 07 CD 07 65 07 69 07 A0 07 81 07 7F 07 90 07 ÿ•Ï•e•i••••••
000007CB00: DE 07 BC 07 DF 07 33 08 FF 07 CE 07 70 07 6F 07 ð•%•ß•2•ÿ•Ï•p•o•
000007CB10: 9F 07 82 07 81 07 93 07 DD 07 BC 07 E0 07 34 08 Ÿ•„••“•ÿ•%•à•4•
000007CB20: FE 07 CE 07 7E 07 78 07 9F 07 84 07 84 07 96 07 þ•Ï•~•x•ÿ•„•„•••-•
000007CB30: DE 07 BD 07 DF 07 32 08 FF 07 CE 07 87 07 7F 07 ð•%•ß•2•ÿ•Ï•†•Δ•
000007CB40: A1 07 87 07 88 07 9B 07 E2 07 BF 07 DE 07 2F 08 j•†•~•>•â•¿•p•/•
000007CB50: 02 08 CF 07 93 07 89 07 A4 07 8C 07 8D 07 9F 07 ø•Ï•“••••••••ÿ•
000007CB60: E4 07 C0 07 DD 07 2D 08 03 08 CF 07 9C 07 92 07 ä•À•ÿ•-•••Ï•e•’•
000007CB70: A9 07 90 07 91 07 A3 07 E6 07 C3 07 DD 07 2B 08 ••••‘••f•æ•Ã•ÿ•+•
000007CB80: 04 08 D0 07 A7 07 9C 07 AE 07 96 07 96 07 A7 07 ♦•D•§•æ•@•-•-•§•
000007CB90: E8 07 C7 07 DF 07 29 08 04 08 D3 07 B1 07 A7 07 è•Ç•ß•)•••Ö•±•§•
000007CBA0: B4 07 9B 07 9B 07 AB 07 E8 07 CA 07 E1 07 27 08 ˆ•>•>•«•è•Ë•á•’•
000007CBB0: 03 08 D5 07 BB 07 B3 07 BB 07 A1 07 A0 07 AF 07 ♥•Ö•»•³•»•j•••ˆ•
000007CBC0: EA 07 CD 07 E3 07 25 08 03 08 D8 07 C4 07 BD 07 è•Ï•ä•%•••Ø•Ä•%•
000007CBD0: C1 07 A6 07 A5 07 B3 07 EA 07 D1 07 E6 07 22 08 Á•‡•¥•³•è•Ñ•æ•"•
000007CBE0: 01 08 DC 07 CE 07 C8 07 C8 07 AD 07 AA 07 B7 07 ø•Û•Ï•Ë•È•-•á•••
1Help 2Wrap 3Quit 4Text 5 6Edit 7Search 80EM 9 10Quit
```

Рис. 5.8: FAR: массив 16-битных значений

Для примера я использовал файл содержащий 12-канальный сигнал оцифрованный при помощи 16-битного **ADC!**<sup>24</sup>.

<sup>24</sup>**ADC!**





## Разреженные файлы

Это разреженный файл, в котором данные разбросаны посреди почти пустого файла. Каждый символ пробела здесь на самом деле нулевой байт (который выглядит как пробел). Это файл для программирования FPGA (чип Altera Stratix GX). Конечно, такие файлы легко сжимаются, но подобные форматы очень популярны в научном и инженерном ПО, где быстрый доступ важен, а компактность — не очень.

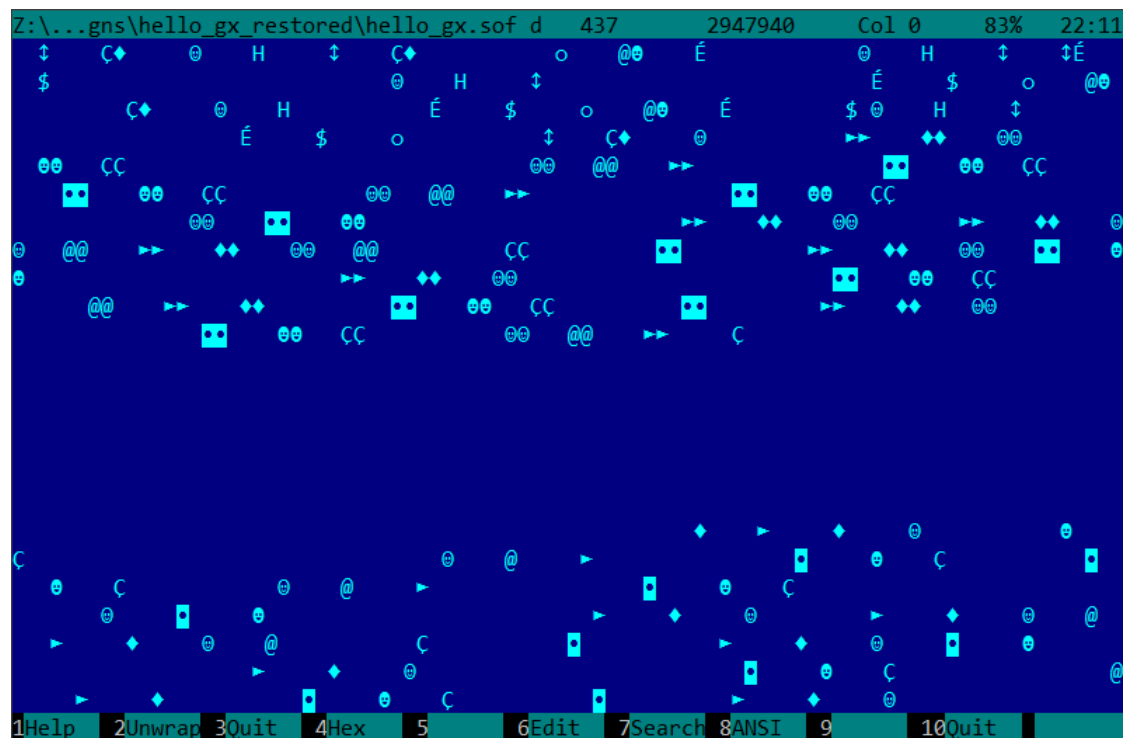


Рис. 5.10: FAR: Разреженный файл

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



**CDFS!**<sup>25</sup>

Инсталляции **OS!** обычно распространяются в ISO-файлах, которые суть копии CD/DVD-дисков. Используемая файловая система называется **CDFS!**, здесь видны имена файлов и какие-то дополнительные данные. Это могут быть длины файлов, указатели на другие директории, атрибуты файлов, и т. д. Так может выглядеть типичная файловая система внутри.

```
Z:\..\untu\ubuntu-15.10-desktop-i386.iso d 437 1226964992 Col 0 0% 21:59
ä # #  s; ; @ @ SP•n
PX$mA Am @ TF→s; , s; , CEL$ $ φ
φ ` # #  s; ; @ @ PX$mA Am @ TF→s; , s;
s; , s; , n % %  s; ; @ @ .DISKPX$mA Am @ TF
→s; , s; , NM .diskn & &  s; ; @ @ BOOT PX$mA Am
@ TF→s; ( s; , s; ( NM boot r ( (  s; ; @ @
↑CASPER PX$mA Am @ TF→s; ( s; , s; ( NM casper n
)  s; ; @ @ DISTSPX$mA Am @ TF→s; → s
s; → NM distr 1 1  s; ; @ @ INSTALLPX$mA Am @
TF→s; ( s; , s; ( NM installv 2 2 H H s; ; @ @ ISOLINUX PX$m
mA Am @ TF→s; ( s; ( s; ( NM isolinux z p p`
s; ; @ MD5SUM.TXT PX$ü ü$ @ TF→s; : s; ; s;
: NM md5sum.txt n ; ;  s; ; @ @ PICS PX$mA Am @
TF→s; → s; → s; → NM pics n < <  s; ; @ @ POOL PX$mA
Am @ TF→s; → s; → s; → NM pool r W W  s; ; @
@ •PRESEEDPX$mA Am @ TF→s; → s; → s; → NM preseed
è S @ Sa as; ; @ $README.DISKDEFINES PX$ü ü$ @
TF→s; → s; → s; → NM README.diskdefines x ≤ ≤ s; ; @ @ UBUNT
U.PX$mí ím @ TF→s; → s; → s; → NM ubuntuSL• @
1Help 2Unwrap 3Quit 4Hex 5 6Edit 7Search 8ANSI 9 10Quit
```

Рис. 5.12: FAR: ISO-файл: инсталляционный **CD!**<sup>26</sup> Ubuntu 15<sup>25</sup> **CDFS!**Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

32-битный x86 исполняемый код

Так выглядит 32-битный x86 исполняемый код. У него не очень высокая энтропия, потому что некоторые байты встречаются чаще других.



Рис. 5.13: FAR: Исполняемый 32-битных x86 код

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

### Графические BMP-файлы

BMP-файлы не сжаты, так что каждый байт (или группа байт) описывают каждый пиксель. Я нашел эту картинку где-то внутри заинсталлированной Windows 8.1:



Рис. 5.14: Пример картинки

Вы видите, что эта картинка имеет пиксели, которые вряд ли могут быть хорошо сжаты (в районе центра), но здесь есть длинные одноцветные линии вверху и внизу. Действительно, линии вроде этих выглядят как линии при просмотре этого файла:

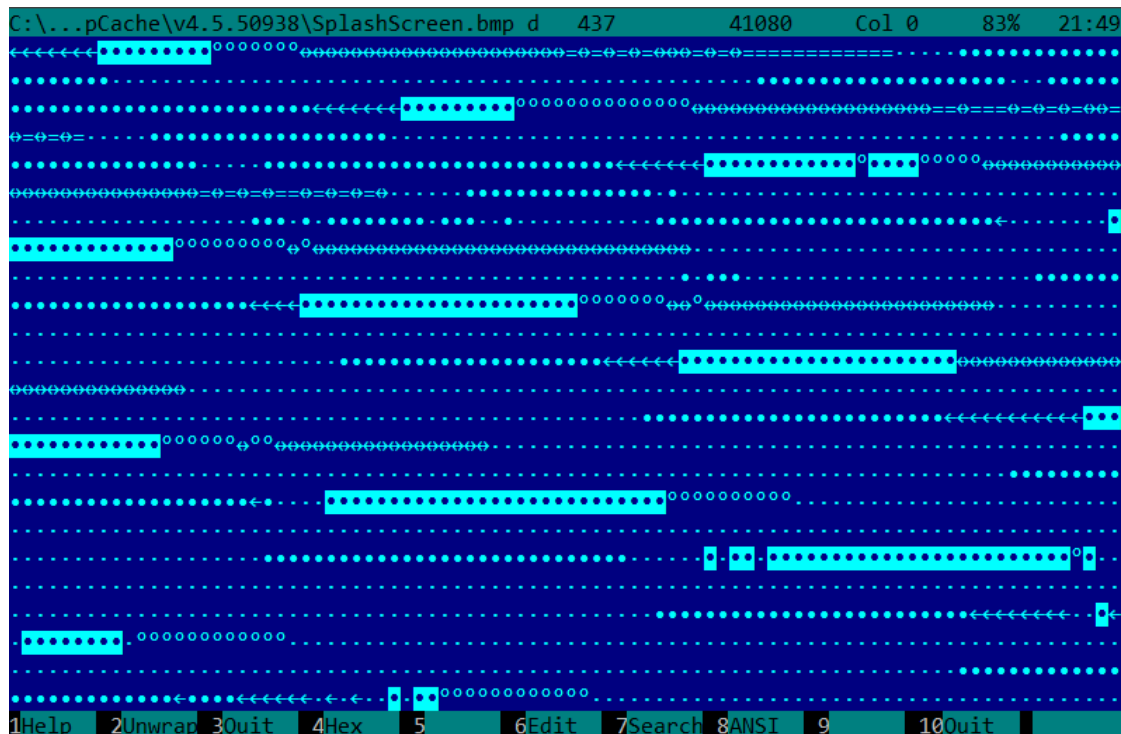


Рис. 5.15: Фрагмент BMP-файла

### 5.10.2. Сравнение «снимков» памяти

Метод простого сравнения двух снимков памяти для поиска изменений часто применялся для взлома игр на 8-битных компьютерах и взлома файлов с записанными рекордными очками.

К примеру, если вы имеете загруженную игру на 8-битном компьютере (где самой памяти не очень много, но игра занимает еще меньше), и вы знаете что сейчас у вас, условно, 100 пуль, вы можете сделать «снимок» всей памяти и сохранить где-то. Затем просто стреляете куда угодно, у вас станет 99 пуль, сделать второй «снимок», и затем сравнить эти два снимка: где-то наверняка должен быть байт, который в начале был 100, а затем стал 99.

Если учесть, что игры на тех маломощных домашних компьютерах обычно были написаны на ассемблере и подобные переменные там были глобальные, то можно с уверенностью сказать, какой адрес в памяти всегда отвечает за количество пуль. Если поискать в дизассемблированном коде игры все обращения по этому адресу, несложно найти код, отвечающий за уменьшение пуль и записать туда инструкцию `NOP` или несколько `NOP`-в, так мы получим игру в которой у игрока всегда будет 100 пуль, например.

А так как игры на тех домашних 8-битных компьютерах всегда загружались по одним и тем же адресам, и версий одной игры редко когда было больше

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

одной продолжительное время, то геймеры-энтузиасты знали, по какому адресу (используя инструкцию языка BASIC **POKE**) что записать после загрузки игры, чтобы хакнуть её. Это привело к появлению списков «читов» состоящих из инструкций **POKE**, публикуемых в журналах посвященным 8-битным играм.

Точно так же легко модифицировать файлы с сохраненными рекордами (кто сколько очков набрал), впрочем, это может сработать не только с 8-битными играми. Нужно заметить, какой у вас сейчас рекорд и где-то сохранить файл с очками. Затем, когда очков станет другое количество, просто сравнить два файла, можно даже DOS-утилитой FC<sup>27</sup> (файлы рекордов, часто, бинарные).

Где-то будут отличаться несколько байт, и легко будет увидеть, какие именно отвечают за количество очков. Впрочем, разработчики игр полностью осведомлены о таких хитростях и могут защититься от этого.

В каком-то смысле похожий пример в этой книге здесь: ?? (стр. ??).

### Реальная история из 1999

В то время был популярен мессенджер ICQ, по крайней мере, в странах бывшего СССР. У мессенджера была особенность — некоторые пользователи не хотели, чтобы все знали, в онлайн они или нет. И для начала у того пользователя нужно было запросить *авторизацию*. Тот человек мог разрешить вам видеть свой статус, а мог и не разрешить.

Автор сих строк сделал следующее.

- Добавил человека. Он появился в контакт-листе, в разделе “wait for authorization”.
- Выгрузил ICQ.
- Сохранил базу ICQ в другом месте.
- Загрузил ICQ снова.
- Человек *авторизировал*.
- Выгрузил ICQ и сравнил две базы.

Выяснилось: базы отличались только одним байтом. В первой версии: RESU\х03, во второй RESU\х02. (“RESU”, надо думать, означало “USER”, т.е., заголовок структуры, где хранилась информация о пользователе.) Это означало, что информация об авторизации хранилась не на сервере, а в клиенте. Вероятно, значение 2/3 отражало статус *авторизованности*.

### Реестр Windows

А еще можно вспомнить сравнение реестра Windows до инсталляции программы и после. Это также популярный метод поиска, какие элементы реестра программа использует.

Наверное это причина, почему так популярны shareware-программы для очистки реестра в Windows.

<sup>27</sup>утилита MS-DOS для сравнения двух файлов побайтово

Кстати, вот как сдать реестр в Windows в текстовые файлы:

```
reg export HKLM HKLM.reg
reg export HKCU HKCU.reg
reg export HKCR HKCR.reg
reg export HKU HKU.reg
reg export HKCC HKCC.reg
```

Затем их можно сравнивать используя diff...

### Инженерное ПО, САД-ы, и т. д.

Если некое ПО использует закрытые (проприетарные) файлы, то и тут можно попытаться что-то выяснить. Сохраняете файл. Затем добавили точку, или линию, или еще какой примитив. Сохранили файл, сравнили. Или сдвинули точку в сторону, сохранили файл, сравнили.

### Блинк-компаратор

Сравнение файлов или слепков памяти вообще, немного напоминает блинк-компаратор<sup>28</sup>: устройство, которое раньше использовали астрономы для поиска движущихся небесных объектов.

Блинк-компаратор позволял быстро переключаться между двух отснятых в разное время кадров, и астроном мог увидеть разницу визуально.

Кстати, при помощи блинк-компаратора, в 1930 был открыт Плутон.

## 5.11. Определение ISA!

Часто, вы можете иметь дело с бинарным файлом для неизвестной **ISA!**. Вероятно, простейший способ определить **ISA!** это пробовать разные в IDA, objdump или другом дизассемблере.

Чтобы этого достичь, нужно понимать разницу между некорректно дизассемблированным кодом, и корректно дизассемблированным.

### 5.11.1. Неверно дизассемблированный код

Практикующие reverse engineer-ы часто сталкиваются с неверно дизассемблированным кодом.

#### Дизассемблирование началось в неверном месте (x86)

В отличие от ARM и MIPS (где у каждой инструкции длина или 2 или 4 байта), x86-инструкции имеют переменную длину, так что, любой дизассемблер, начиная работу с середины x86-инструкции, может выдать неверные результаты.

Как пример:

<sup>28</sup><https://ru.wikipedia.org/wiki/Блинк-компаратор>



```

add    [ebp-31F7Bh], cl
dec    dword ptr [ecx-3277Bh]
dec    dword ptr [ebp-2CF7Bh]
inc    dword ptr [ebx-7A76F33Ch]
fdiv   st(4), st
db 0FFh
dec    dword ptr [ecx-21F7Bh]
dec    dword ptr [ecx-22373h]
dec    dword ptr [ecx-2276Bh]
dec    dword ptr [ecx-22B63h]
dec    dword ptr [ecx-22F4Bh]
dec    dword ptr [ecx-23343h]
jmp    dword ptr [esi-74h]
xchg   eax, ebp
clc
std
db 0FFh
db 0FFh
mov    word ptr [ebp-214h], cs ; <- дизассемблер наконец нашел здесь
      правильный старт
mov    word ptr [ebp-238h], ds
mov    word ptr [ebp-23Ch], es
mov    word ptr [ebp-240h], fs
mov    word ptr [ebp-244h], gs
pushf
pop    dword ptr [ebp-210h]
mov    eax, [ebp+4]
mov    [ebp-218h], eax
lea   eax, [ebp+4]
mov    [ebp-20Ch], eax
mov    dword ptr [ebp-2D0h], 10001h
mov    eax, [eax-4]
mov    [ebp-21Ch], eax
mov    eax, [ebp+0Ch]
mov    [ebp-320h], eax
mov    eax, [ebp+10h]
mov    [ebp-31Ch], eax
mov    eax, [ebp+4]
mov    [ebp-314h], eax
call   ds:IsDebuggerPresent
mov    edi, eax
lea   eax, [ebp-328h]
push  eax
call   sub_407663
pop    ecx
test   eax, eax
jnz   short loc_402D7B

```

В начале мы видим неверно дизассемблированные инструкции, но потом, так или иначе, дизассемблер находит верный след.

## Как выглядят случайные данные в дизассемблированном виде?

Общее, что можно сразу заметить, это:

- Необычно большой разброс инструкций. Самые частые x86-инструкции это PUSH, MOV, CALL, но здесь мы видим инструкции из любых групп: **FPU!**-инструкции, инструкции IN/OUT, редкие и системные инструкции, всё друг с другом смешано в одном месте.
- Большие и случайные значения, смещения, immediates.
- Переходы с неверными смещениями часто имеют адрес перехода в середину другой инструкции.

Листинг 5.7: случайный шум (x86)

```

mov     bl, 0Ch
mov     ecx, 0D38558Dh
mov     eax, ds:2C869A86h
db      67h
mov     dl, 0CCh
insb
movsb
push    eax
xor     [edx-53h], ah
fcom   qword ptr [edi-45A0EF72h]
pop     esp
pop     ss
in      eax, dx
dec     ebx
push    esp
lds     esp, [esi-41h]
retf
rcl    dword ptr [eax], cl
mov     cl, 9Ch
mov     ch, 0DFh
push    cs
insb
mov     esi, 0D9C65E4Dh
imul   ebp, [ecx], 66h
pushf
sal    dword ptr [ebp-64h], cl
sub    eax, 0AC433D64h
out    8Ch, eax
pop     ss
sbb    [eax], ebx
aas
xchg   cl, [ebx+ebx*4+14B31Eh]
jecxz  short near ptr loc_58+1
xor    al, 0C6h
inc    edx
db     36h
pusha
stosb
test   [ebx], ebx

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

sub    al, 0D3h ; 'L'
pop    eax
stosb

loc_58: ; CODE XREF: seg000:0000004A
test   [esi], eax
inc    ebp
das
db     64h
pop    ecx
das
hlt

pop    edx
out    0B0h, al
lods b
push   ebx
cdq
out    dx, al
sub    al, 0Ah
sti
outs d
add    dword ptr [edx], 96FCBE4Bh
and    eax, 0E537EE4Fh
inc    esp
stos d
cdq
push   ecx
in     al, 0CBh
mov    ds:0D114C45Ch, al
mov    esi, 659D1985h

```

Листинг 5.8: случайный шум (x86-64)

```

lea    esi, [rax+rdx*4+43558D29h]

loc_AF3: ; CODE XREF: seg000:00000000000000B46
rcl    byte ptr [rsi+rax*8+29BB423Ah], 1
lea    ecx, cs:0FFFFFFFB2A6780Fh
mov    al, 96h
mov    ah, 0CEh
push   rsp
lods   byte ptr [esi]

db    2Fh ; /

pop    rsp
db    64h
retf   0E993h

cmp    ah, [rax+4Ah]
movzx  rsi, dword ptr [rbp-25h]
push   4Ah
movzx  rdi, dword ptr [rdi+rdx*8]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

db 9Ah

rcr    byte ptr [rax+1Dh], cl
lodsd
xor    [rbp+6CF20173h], edx
xor    [rbp+66F8B593h], edx
push  rbx
sbb   ch, [rbx-0Fh]
stosd
int    87h
db     46h, 4Ch
out    33h, rax
xchg  eax, ebp
test  ecx, ebp
movsd
leave
push  rsp

db 16h

xchg  eax, esi
pop   rdi

loc_B3D: ; CODE XREF: seg000:00000000000000B5F
mov   ds:93CA685DF98A90F9h, eax
jnz   short near ptr loc_AF3+6
out   dx, eax
cwde
mov   bh, 5Dh ; '['
movsb
pop   rbp

```

Листинг 5.9: случайный шум (ARM (Режим ARM))

```

BLNE  0xFE16A9D8
BGE   0x1634D0C
SVCCS 0x450685
STRNVT R5, [PC], #-0x964
LDCGE  p6, c14, [R0], #0x168
STCCSL p9, c9, [LR], #0x14C
CMNHIP PC, R10, LSL#22
FLDMIADNV LR!, {D4}
MCR    p5, 2, R2, c15, c6, 4
BLGE  0x1139558
BLGT  0xFF9146E4
STRNEB R5, [R4], #0xCA2
STMNEIB R5, {R0, R4, R6, R7, R9-SP, PC}
STMIA  R8, {R0, R2-R4, R7, R8, R10, SP, LR}^
STRB   SP, [R8], PC, ROR#18
LDCCS  p9, c13, [R6], #0x1BC]
LDRGE  R8, [R9], #0x66E]
STRNEB R5, [R8], #-0x8C3
STCCSL p15, c9, [R7], #-0x84]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

RSBLS LR, R2, R11, ASR LR
SVC GT 0x9B0362
SVC GT 0xA73173
STMNEDB R11!, {R0,R1,R4-R6,R8,R10,R11,SP}
STR R0, [R3], #-0xCE4
LDCGT p15, c8, [R1, #0x2CC]
LDRCCB R1, [R11], -R7, ROR#30
BLLT 0xFED9D58C
BL 0x13E60F4
LDMVSIB R3!, {R1,R4-R7}^
USATNE R10, #7, SP, LSL#11
LDRGEB LR, [R1], #0xE56
STRPLT R9, [LR], #0x567
LDRLT R11, [R1], #-0x29B
SVCNV 0x12DB29
MVNNVS R5, SP, LSL#25
LDCL p8, c14, [R12, #-0x288]
STCNEL p2, c6, [R6, #-0xBC]!
SVCNV 0x2E5A2F
BLX 0x1A8C97E
TEQGE R3, #0x1100000
STMLSIA R6, {R3,R6,R10,R11,SP}
BICPLS R12, R2, #0x5800
BNE 0x7CC408
TEQGE R2, R4, LSL#20
SUBS R1, R11, #0x28C
BICVS R3, R12, R7, ASR R0
LDRMI R7, [LR], R3, LSL#21
BLMI 0x1A79234
STMVCDB R6, {R0-R3,R6,R7,R10,R11}
EORMI R12, R6, #0xC5
MCR RCS p1, 0xF, R1, R3, c2

```

Листинг 5.10: случайный шум (ARM (Режим Thumb))

```

LSRS R3, R6, #0x12
LDRH R1, [R7, #0x2C]
SUBS R0, #0x55 ; 'U'
ADR R1, loc_3C
LDR R2, [SP, #0x218]
CMP R4, #0x86
SXTB R7, R4
LDR R4, [R1, #0x4C]
STR R4, [R4, R2]
STR R0, [R6, #0x20]
BGT 0xFFFFFFFF72
LDRH R7, [R2, #0x34]
LDRSH R0, [R2, R4]
LDRB R2, [R7, R2]

DCB 0x17
DCB 0xED

STRB R3, [R1, R1]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

STR    R5, [R0,#0x6C]
LDMIA  R3, {R0-R5,R7}
ASRS   R3, R2, #3
LDR    R4, [SP,#0x2C4]
SVC    0xB5
LDR    R6, [R1,#0x40]
LDR    R5, =0xB2C5CA32
STMIA  R6, {R1-R4,R6}
LDR    R1, [R3,#0x3C]
STR    R1, [R5,#0x60]
BCC    0xFFFFFFFF70
LDR    R4, [SP,#0x1D4]
STR    R5, [R5,#0x40]
ORRS   R5, R7

loc_3C ; DATA XREF: ROM:00000006
B      0xFFFFFFFF98

```

Листинг 5.11: случайный шум (MIPS little endian)

```

lw     $t9, 0xCB3($t5)
sb     $t5, 0x3855($t0)
sltiu  $a2, $a0, -0x657A
ldr    $t4, -0x4D99($a2)
daddi  $s0, $s1, 0x50A4
lw     $s7, -0x2353($s4)
bgtzl  $a1, 0x17C5C

.byte 0x17
.byte 0xED
.byte 0x4B # K
.byte 0x54 # T

lwc2   $31, 0x66C5($sp)
lwu    $s1, 0x10D3($a1)
ldr    $t6, -0x204B($zero)
lwc1   $f30, 0x4DBE($s2)
daddiu $t1, $s1, 0x6BD9
lwu    $s5, -0x2C64($v1)
cop0   0x13D642D
bne    $gp, $t4, 0xFFFF9EF0
lh     $ra, 0x1819($s1)
sdl    $fp, -0x6474($t8)
jal    0x78C0050
ori    $v0, $s2, 0xC634
blez   $gp, 0xFFFEA9D4
swl    $t8, -0x2CD4($s2)
sltiu  $a1, $k0, 0x685
sdc1   $f15, 0x5964($at)
sw     $s0, -0x19A6($a1)
sltiu  $t6, $a3, -0x66AD
lb     $t7, -0x4F6($t3)
sd     $fp, 0x4B02($a1)

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Также важно помнить, что хитрым образом написанный код для распаковки и дешифровки (включая саомодифицирующийся), также может выглядеть как случайный шум, тем не менее, он исполняется корректно.

### 5.11.2. Корректно дизассемблированный код

Каждая **ISA!** имеет десяток самых используемых инструкций, остальные используются куда реже.

Интересно знать тот факт, что в x86, инструкции вызовов ф-ций (PUSH/CALL/ADD) и MOV это наиболее часто исполняющиеся инструкции в коде почти во всем ПО что мы используем. Другими словами, **CPU!** очень занят передачей информации между уровнями абстракции, или, можно сказать, очень занят переключением между этими уровнями. Вне зависимости от **ISA!**. Это цена расслоения программ на разные уровни абстракций (чтобы человеку было легче с ними управляться).

## 5.12. Прочее

### 5.12.1. Общая идея

Нужно стараться как можно чаще ставить себя на место программиста и задавать себе вопрос, как бы вы сделали ту или иную вещь в этом случае и в этой программе.

### 5.12.2. Порядок функций в бинарном коде

Все функции расположенные в одном .с или .cpp файле компилируются в соответствующий объектный (.o) файл. Линкер впоследствии складывает все нужные объектные файлы вместе, не меняя порядок ф-ций в них. Как следствие, если вы видите в коде две или более идущих подряд ф-ций, то это означает, что и в исходном коде они были расположены в одном и том же файле (если только вы не на границе двух объектных файлов, конечно). Это может означать, что эти ф-ции имеют что-то общее между собой, что они из одного слоя **API!**, из одной библиотеки, и т. д.

Это реальная история из практики: однажды автор искал в прикомпилированной библиотеке CryptoPP ф-ции связанные с алгоритмом Twofish, особенно шифрования/дешифрования.

Я нашел ф-цию Twofish::Base::UncheckedSetKey(), но не остальные. Заглянув в исходники twofish.cpp<sup>29</sup>, стало ясно, что все ф-ции расположены в одном модуле (twofish.cpp). Так что я просто попробовал посмотреть ф-ции следующие за

Twofish::Base::UncheckedSetKey() — так и оказалось, одна из них была Twofish::Enc::ProcessAndXorBlock(), другая — Twofish::Dec::ProcessAndXorBlock().

<sup>29</sup><https://github.com/weidai11/cryptopp/blob/b613522794a7633aa2bd81932a98a0b0a51bc04f/twofish.cpp>

### 5.12.3. Крохотные функции

Крохотные ф-ции, такие как пустые ф-ции (?? (стр. ??)) или ф-ции возвращающие только “true” (1) или “false” (0) (?? (стр. ??)) очень часто встречаются, и почти все современные компиляторы, как правило, помещают только одну такую ф-цию в исполняемый код, даже если в исходном их было много одинаковых. Так что если вы видите ф-цию состоящую только из `mov eax, 1 / ret`, которая может вызываться из разных мест, которые, судя по всему, друг с другом никак не связаны, это может быть результат подобной оптимизации.

### 5.12.4. Си++

**RTTI!** (?? (стр. ??))-информация также может быть полезна для идентификации классов в Си++.

### 5.12.5. Намеренный сбой

Часто, нужно знать, какая ф-ция была исполнена, а какая — нет. Вы можете использовать отладчик, но на экзотических архитектурах его может и не быть, так что простейший способ это вписать туда неверный опкод или что-то вроде INT3 (0xCC). Сбой будет сигнализировать о том, что эта инструкция была исполнена.

Еще один пример намеренного сбоя: ?? (стр. ??).



## Глава 6

# Специфичное для ОС

### 6.1. Способы передачи аргументов при вызове функций

#### 6.1.1. cdecl

Этот способ передачи аргументов через стек чаще всего используется в языках Си/Си++.

Вызывающая функция заталкивает в стек аргументы в обратном порядке: сначала последний аргумент в стек, затем предпоследний, и в самом конце — первый аргумент. Вызывающая функция должна также затем вернуть [указатель стека](#) в нормальное состояние, после возврата вызываемой функции.

Листинг 6.1: cdecl

```
push arg3
push arg2
push arg1
call function
add esp, 12 ; возвращает ESP
```

#### 6.1.2. stdcall

Это почти то же что и *cdecl*, за исключением того, что вызываемая функция сама возвращает ESP в нормальное состояние, выполнив инструкцию RET *x* вместо RET,

где *x* = количество\_аргументов \* sizeof(int)<sup>1</sup>. Вызывающая функция не будет корректировать [указатель стека](#), там нет инструкции add esp, *x*.

Листинг 6.2: stdcall

```
push arg3
```

<sup>1</sup>Размер переменной типа *int*— 4 в x86-системах и 8 в x64-системах

```
push arg2
push arg1
call function
```

```
function:
;... сделать что-то ...
ret 12
```

Этот способ используется почти везде в системных библиотеках win32, но не в win64 (о win64 смотрите ниже).

Например, мы можем взять функцию из ?? (стр. ??) и изменить её немного добавив модификатор `__stdcall`:

```
int __stdcall f2 (int a, int b, int c)
{
    return a*b+c;
};
```

Он будет скомпилирован почти так же как и ?? (стр. ??), но вы увидите `RET 12` вместо `RET`. **SP!** не будет корректироваться в [вызывающей функции](#).

Как следствие, количество аргументов функции легко узнать из инструкции `RETN n` просто разделите  $n$  на 4.

#### Листинг 6.3: MSVC 2010

```
_a$ = 8      ; size = 4
_b$ = 12     ; size = 4
_c$ = 16     ; size = 4
_f2@12 PROC
    push     ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    imul   eax, DWORD PTR _b$[ebp]
    add    eax, DWORD PTR _c$[ebp]
    pop     ebp
    ret     12
_f2@12 ENDP
; ...
    push    3
    push    2
    push    1
    call    _f2@12
    push    eax
    push    OFFSET $SG81369
    call    _printf
    add    esp, 8
```

### Функции с переменным количеством аргументов

Функции вроде `printf()`, должно быть, единственный случай функций в Си/Си++ с переменным количеством аргументов, но с их помощью можно легко

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

проследить очень важную разницу между *cdecl* и *stdcall*. Начнем с того, что компилятор знает сколько аргументов было у `printf()`.

Однако, вызываемая функция `printf()`, которая уже давно скомпилирована и находится в системной библиотеке `MSVCRT.DLL` (если говорить о Windows), не знает сколько аргументов ей передали, хотя может установить их количество по строке формата.

Таким образом, если бы `printf()` была *stdcall*-функцией и возвращала [указатель стека](#) в первоначальное состояние подсчитав количество аргументов в строке формата, это была бы потенциально опасная ситуация, когда одна опечатка программиста могла бы вызывать неожиданные падения программы. Таким образом, для таких функций *stdcall* явно не подходит, а подходит *cdecl*.

### 6.1.3. fastcall

Это общее название для передачи некоторых аргументов через регистры, а всех остальных — через стек. На более старых процессорах, это работало потенциально быстрее чем *cdecl/stdcall* (ведь стек в памяти использовался меньше). Впрочем, на современных (намного более сложных) CPU, существенного выигрыша может и не быть.

Это не стандартизированный способ, поэтому разные компиляторы делают это по-своему. Разумеется, если у вас есть, скажем, две DLL, одна использует другую, и обе они собраны с *fastcall* но разными компиляторами, очень вероятно, будут проблемы.

MSVC и GCC передает первый и второй аргумент через ECX и EDX а остальные аргументы через стек.

[Указатель стека](#) должен быть возвращен в первоначальное состояние вызываемой функцией, как в случае *stdcall*.

Листинг 6.4: *fastcall*

```
push arg3
mov edx, arg2
mov ecx, arg1
call function

function:
; .. делать что-то ..
ret 4
```

Например, мы можем взять функцию из ?? (стр. ??) и изменить её немного добавив модификатор `__fastcall`:

```
int __fastcall f3 (int a, int b, int c)
{
    return a*b+c;
};
```

Вот как он будет скомпилирован:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Листинг 6.5: Оптимизирующий MSVC 2010 /Ob0

```

_c$ = 8           ; size = 4
@f3@12 PROC
; _a$ = ecx
; _b$ = edx
        mov     eax, ecx
        imul   eax, edx
        add    eax, DWORD PTR _c$[esp-4]
        ret    4
@f3@12 ENDP

; ...

        mov     edx, 2
        push   3
        lea   ecx, DWORD PTR [edx-1]
        call  @f3@12
        push  eax
        push  OFFSET $SG81390
        call  _printf
        add   esp, 8

```

Видно, что **вызываемая функция** сама возвращает **SP!** при помощи инструкции RETN с операндом. Так что и здесь можно легко вычислять количество аргументов.

### GCC regparm

Это в некотором роде, развитие *fastcall*<sup>2</sup>. Опцией `-mregparm=x` можно указывать, сколько аргументов компилятор будет передавать через регистры. Максимально 3. В этом случае будут задействованы регистры EAX, EDX и ECX.

Разумеется, если аргументов у функции меньше трех, то будет задействована только часть регистров.

Вызывающая функция возвращает **указатель стека** в первоначальное состояние.

Для примера, см. (?? (стр. ??)).

### Watcom/OpenWatcom

Здесь это называется «register calling convention». Первые 4 аргумента передаются через регистры EAX, EDX, EBX and ECX. Все остальные — через стек. Эти функции имеют символ подчеркивания, добавленный к концу имени функции, для отличия их от тех, которые имеют другой способ передачи аргументов.

## 6.1.4. thiscall

В Си++, это передача в функцию-метод указателя *this* на объект.

<sup>2</sup><http://www.ohse.de/uwe/articles/gcc-attributes.html#func-regparm>

В MSVC указатель *this* обычно передается в регистре ECX.

В GCC указатель *this* обычно передается как самый первый аргумент. Таким образом, в коде на ассемблере будет видно: у всех функций-методов на один аргумент больше, чем в исходном коде.

Для примера, см. (?? (стр. ??)).

### 6.1.5. x86-64

#### Windows x64

В win64 метод передачи всех параметров немного похож на `fastcall`. Первые 4 аргумента записываются в регистры RCX, RDX, R8, R9, а остальные — в стек. Вызывающая функция также должна подготовить место из 32 байт или для четырех 64-битных значений, чтобы вызываемая функция могла сохранить там первые 4 аргумента. Короткие функции могут использовать переменные прямо из регистров, но большие могут сохранять их значения на будущее.

Вызывающая функция должна вернуть [указатель стека](#) в первоначальное состояние.

Это же соглашение используется и в системных библиотеках Windows x86-64 (вместо `stdcall` в win32).

Пример:

```
#include <stdio.h>

void f1(int a, int b, int c, int d, int e, int f, int g)
{
    printf ("%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g);
};

int main()
{
    f1(1,2,3,4,5,6,7);
};
```

Листинг 6.6: MSVC 2012 /Ob

```
$SG2937 DB      '%d %d %d %d %d %d %d', 0aH, 00H

main  PROC
      sub     rsp, 72

      mov     DWORD PTR [rsp+48], 7
      mov     DWORD PTR [rsp+40], 6
      mov     DWORD PTR [rsp+32], 5
      mov     r9d, 4
      mov     r8d, 3
      mov     edx, 2
      mov     ecx, 1
      call    f1
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        xor    eax, eax
        add    rsp, 72
        ret    0
main    ENDP

a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1     PROC
$LN3:
        mov    DWORD PTR [rsp+32], r9d
        mov    DWORD PTR [rsp+24], r8d
        mov    DWORD PTR [rsp+16], edx
        mov    DWORD PTR [rsp+8], ecx
        sub    rsp, 72

        mov    eax, DWORD PTR g$[rsp]
        mov    DWORD PTR [rsp+56], eax
        mov    eax, DWORD PTR f$[rsp]
        mov    DWORD PTR [rsp+48], eax
        mov    eax, DWORD PTR e$[rsp]
        mov    DWORD PTR [rsp+40], eax
        mov    eax, DWORD PTR d$[rsp]
        mov    DWORD PTR [rsp+32], eax
        mov    r9d, DWORD PTR c$[rsp]
        mov    r8d, DWORD PTR b$[rsp]
        mov    edx, DWORD PTR a$[rsp]
        lea   rcx, OFFSET FLAT:$SG2937
        call  printf

        add    rsp, 72
        ret    0
f1     ENDP

```

Здесь мы легко видим, как 7 аргументов передаются: 4 через регистры и остальные 3 через стек. Код пролога функции f1() сохраняет аргументы в «scratch space» — место в стеке предназначенное именно для этого. Это делается потому что компилятор может быть не уверен, достаточно ли ему будет остальных регистров для работы исключая эти 4, которые иначе будут заняты аргументами до конца исполнения функции. Выделение «scratch space» в стеке лежит на ответственности вызывающей функции.

#### Листинг 6.7: Оптимизирующий MSVC 2012 /Ob

```

$SG2777 DB    '%d %d %d %d %d %d %d', 0aH, 00H

a$ = 80
b$ = 88

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1      PROC
$LN3:
        sub     rsp, 72

        mov     eax, DWORD PTR g$[rsp]
        mov     DWORD PTR [rsp+56], eax
        mov     eax, DWORD PTR f$[rsp]
        mov     DWORD PTR [rsp+48], eax
        mov     eax, DWORD PTR e$[rsp]
        mov     DWORD PTR [rsp+40], eax
        mov     DWORD PTR [rsp+32], r9d
        mov     r9d, r8d
        mov     r8d, edx
        mov     edx, ecx
        lea    rcx, OFFSET FLAT:$SG2777
        call   printf

        add     rsp, 72
        ret     0
f1      ENDP
main    PROC
        sub     rsp, 72

        mov     edx, 2
        mov     DWORD PTR [rsp+48], 7
        mov     DWORD PTR [rsp+40], 6
        lea    r9d, QWORD PTR [rdx+2]
        lea    r8d, QWORD PTR [rdx+1]
        lea    ecx, QWORD PTR [rdx-1]
        mov     DWORD PTR [rsp+32], 5
        call   f1

        xor     eax, eax
        add     rsp, 72
        ret     0
main    ENDP

```

Если компилировать этот пример с оптимизацией, то выйдет почти то же самое, только «scratch space» не используется, потому что незачем.

Обратите также внимание на то как MSVC 2012 оптимизирует примитивную загрузку значений в регистры используя LEA (?? (стр. ??)). MOV здесь был бы на 1 байт длиннее (5 вместо 4).

Еще один пример подобного: ?? (стр. ??).

## Windows x64: Передача *this* (Си/Си++)

Указатель *this* передается через RCX, первый аргумент метода через RDX, и т. д. Для примера, см. также: ?? (стр. ??).

## Linux x64

Метод передачи аргументов в Linux для x86-64 почти такой же, как и в Windows, но 6 регистров используется вместо 4 (RDI, RSI, RDX, RCX, R8, R9), и здесь нет «scratch space», но *callee* может сохранять значения регистров в стеке, если ему это нужно.

Листинг 6.8: Оптимизирующий GCC 4.7.3

```
.LC0:
    .string "%d %d %d %d %d %d %d\n"
f1:
    sub    rsp, 40
    mov    eax, DWORD PTR [rsp+48]
    mov    DWORD PTR [rsp+8], r9d
    mov    r9d, ecx
    mov    DWORD PTR [rsp], r8d
    mov    ecx, esi
    mov    r8d, edx
    mov    esi, OFFSET FLAT:.LC0
    mov    edx, edi
    mov    edi, 1
    mov    DWORD PTR [rsp+16], eax
    xor    eax, eax
    call   __printf_chk
    add    rsp, 40
    ret
main:
    sub    rsp, 24
    mov    r9d, 6
    mov    r8d, 5
    mov    DWORD PTR [rsp], 7
    mov    ecx, 4
    mov    edx, 3
    mov    esi, 2
    mov    edi, 1
    call   f1
    add    rsp, 24
    ret
```

Н.В.: здесь значения записываются в 32-битные части регистров (например EAX) а не в весь 64-битный регистр (RAX). Это связано с тем что в x86-64, запись в младшую 32-битную часть 64-битного регистра автоматически обнуляет старшие 32 бита. Должно быть, это так решили в AMD для упрощения портирования кода под x86-64.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



### 6.1.6. Возвращение переменных типа *float*, *double*

Во всех соглашениях кроме Win64, переменная типа *float* или *double* возвращается через регистр FPU ST(0).

В Win64 переменные типа *float* и *double* возвращаются в младших 16-и или 32-х битах регистра XMM0.

### 6.1.7. Модификация аргументов

Иногда программисты на Си/Си++ (и не только этих **PL!**) задаются вопросом, что может случиться, если модифицировать аргументы?

Ответ прост: аргументы хранятся в стеке, именно там и будет происходить модификация.

А вызывающие функции не используют их после вызова функции (автор никогда не видел в своей практике обратного случая).

```
#include <stdio.h>

void f(int a, int b)
{
    a=a+b;
    printf ("%d\n", a);
};
```

Листинг 6.9: MSVC 2012

```
_a$ = 8 ; size = 4
_b$ = 12 ; size = 4
_f PROC ; size = 4
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    add     eax, DWORD PTR _b$[ebp]
    mov     DWORD PTR _a$[ebp], eax
    mov     ecx, DWORD PTR _a$[ebp]
    push    ecx
    push    OFFSET $SG2938 ; '%d', 0aH
    call   _printf
    add     esp, 8
    pop     ebp
    ret     0
_f ENDP
```

Следовательно, модифицировать аргументы функции можно запросто. Разумеется, если это не *references* в Си++ (**??** (стр. **??**)), и если вы не модифицируете данные по указателю, то эффект не будет распространяться за пределами текущей функции.

Теоретически, после возврата из *callee*, функция-*caller* могла бы получить модифицированный аргумент и использовать его как-то. Может быть, если бы она была написана на языке ассемблера.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Например, такой код генерирует обычный компилятор Си/Си++:

```
push    456      ; будет b
push    123      ; будет a
call    f        ; f() модифицирует свой первый аргумент
add     esp, 2*4
```

Мы можем переписать так:

```
push    456      ; будет b
push    123      ; будет a
call    f        ; f() модифицирует свой первый аргумент
pop     eax
add     esp, 4
; EAX=1-й аргумент f() модифицированный в f()
```

Трудно представить, кому может это понадобиться, но на практике это возможно. Так или иначе, стандарты языков Си/Си++ не предлагают никакого способа это сделать.

### 6.1.8. Указатель на аргумент функции

...и даже более того, можно взять указатель на аргумент функции и передать его в другую функцию:

```
#include <stdio.h>

// located in some other file
void modify_a (int *a);

void f (int a)
{
    modify_a (&a);
    printf ("%d\n", a);
};
```

Трудно понять, как это работает, пока мы не посмотрим на код:

Листинг 6.10: Оптимизирующий MSVC 2010

```
$SG2796 DB      '%d', 0aH, 00H

_a$ = 8
_f      PROC
    lea    eax, DWORD PTR _a$[esp-4] ; just get the address of value in
    local stack
    push   eax                        ; and pass it to modify_a()
    call   _modify_a
    mov    ecx, DWORD PTR _a$[esp]   ; reload it from the local stack
    push   ecx                        ; and pass it to printf()
    push   OFFSET $SG2796            ; '%d'
    call   _printf
    add    esp, 12
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    ret    0
_f     ENDP

```

Адрес места в стеке где была передана *a* просто передается в другую функцию. Она модифицирует переменную по этому адресу, и затем `printf()` выведет модифицированное значение.

Наблюдательный читатель может спросить, а что насчет тех соглашений о вызовах, где аргументы функции передаются в регистрах?

Это та самая ситуация, где используется *Shadow Space*.

Так что входящее значение копируется из регистра в *Shadow Space* в локальном стеке и затем этот адрес передается в другую функцию:

Листинг 6.11: Оптимизирующий MSVC 2012 x64

```

$SG2994 DB    '%d', 0aH, 00H

a$ = 48
f         PROC
        mov     DWORD PTR [rsp+8], ecx    ; save input value in Shadow Space
        sub     rsp, 40
        lea    rcx, QWORD PTR a$[rsp]    ; get address of value and pass it
        to modify_a()
        call   modify_a
        mov     edx, DWORD PTR a$[rsp]    ; reload value from Shadow Space and
        pass it to printf()
        lea    rcx, OFFSET FLAT:$SG2994 ; '%d'
        call   printf
        add    rsp, 40
        ret     0
f         ENDP

```

GCC также записывает входное значение в локальный стек:

Листинг 6.12: Оптимизирующий GCC 4.9.1 x64

```

.LC0:
    .string "%d\n"
f:
    sub     rsp, 24
    mov     DWORD PTR [rsp+12], edi    ; store input value to the local
    stack
    lea    rdi, [rsp+12]                ; take an address of the value and
    pass it to modify_a()
    call   modify_a
    mov     edx, DWORD PTR [rsp+12]    ; reload value from the local stack
    and pass it to printf()
    mov     esi, OFFSET FLAT:.LC0     ; '%d'
    mov     edi, 1
    xor     eax, eax
    call   __printf_chk
    add    rsp, 24
    ret

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

GCC для ARM64 делает то же самое, но это пространство здесь называется *Register Save Area*:

Листинг 6.13: Оптимизирующий GCC 4.9.1 ARM64

```
f:
    stp    x29, x30, [sp, -32]!
    add    x29, sp, 0           ; setup FP
    add    x1, x29, 32         ; calculate address of variable in
Register Save Area
    str    w0, [x1,-4]!       ; store input value there
    mov    x0, x1             ; pass address of variable to the
modify_a()
    bl    modify_a
    ldr    w1, [x29,28]       ; load value from the variable and pass it
to printf()
    adrp   x0, .LC0           ; '%d'
    add    x0, x0, :lo12:LC0
    bl    printf              ; call printf()
    ldp    x29, x30, [sp], 32
    ret
.LC0:
    .string "%d\n"
```

Кстати, похожее использование *Shadow Space* разбирается здесь: ?? (стр. ??).

## 6.2. Thread Local Storage

Это область данных, отдельная для каждого треда. Каждый тред может хранить там то, что ему нужно. Один из известных примеров, это стандартная глобальная переменная в Си *errno*. Несколько тредов одновременно могут вызывать функции возвращающие код ошибки в *errno*, поэтому глобальная переменная здесь не будет работать корректно, для мультитредовых программ *errno* нужно хранить в **TLS!**.

В C++11 ввели модификатор *thread\_local*, показывающий, что каждый тред будет иметь свою версию этой переменной, и её можно инициализировать, и она расположена в **TLS!**<sup>3</sup>:

Листинг 6.14: C++11

```
#include <iostream>
#include <thread>

thread_local int tmp=3;

int main()
{
    std::cout << tmp << std::endl;
};
```

<sup>3</sup> В C11 также есть поддержка тредов, хотя и опциональная

Компилируется в MinGW GCC 4.8.1, но не в MSVC 2012.

Если говорить о PE-файлах, то в исполняемом файле значение *tmp* будет размещено именно в секции отведенной **TLS!**

### 6.2.1. Вернемся к линейному конгруэнтному генератору

Рассмотренный ранее ?? (стр. ??) генератор псевдослучайных чисел имеет недостаток: он не пригоден для многопоточной среды, потому что переменная его внутреннего состояния может быть прочитана и/или модифицирована в разных потоках одновременно.

#### Win32

#### Неинициализированные данные в TLS!

Одно из решений — это добавить модификатор `__declspec( thread )` к глобальной переменной, и теперь она будет выделена в **TLS!** (строка 9):

```

1 #include <stdint.h>
2 #include <windows.h>
3 #include <winnt.h>
4
5 // from the Numerical Recipes book:
6 #define RNG_a 1664525
7 #define RNG_c 1013904223
8
9 __declspec( thread ) uint32_t rand_state;
10
11 void my_srand (uint32_t init)
12 {
13     rand_state=init;
14 }
15
16 int my_rand ()
17 {
18     rand_state=rand_state*RNG_a;
19     rand_state=rand_state+RNG_c;
20     return rand_state & 0x7fff;
21 }
22
23 int main()
24 {
25     my_srand(0x12345678);
26     printf ("%d\n", my_rand());
27 };

```

Niew показывает что в исполняемом файле теперь есть новая PE-секция: `.tls`.

Листинг 6.15: Оптимизирующий MSVC 2013 x86

<code>__TLS</code>	SEGMENT
--------------------	---------

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

_rand_state DD 01H DUP (?)
_TLS      ENDS

_DATA    SEGMENT
$SG84851 DB    '%d', 0aH, 00H
_DATA    ENDS
_TEXT    SEGMENT

_init$ = 8      ; size = 4
_my_srand PROC
; FS:0=address of TIB
    mov     eax, DWORD PTR fs:__tls_array ; displayed in IDA as FS:2Ch
; EAX=address of TLS of process
    mov     ecx, DWORD PTR __tls_index
    mov     ecx, DWORD PTR [eax+ecx*4]
; ECX=current TLS segment
    mov     eax, DWORD PTR _init$[esp-4]
    mov     DWORD PTR _rand_state[ecx], eax
    ret     0
_my_srand ENDP

_my_rand PROC
; FS:0=address of TIB
    mov     eax, DWORD PTR fs:__tls_array ; displayed in IDA as FS:2Ch
; EAX=address of TLS of process
    mov     ecx, DWORD PTR __tls_index
    mov     ecx, DWORD PTR [eax+ecx*4]
; ECX=current TLS segment
    imul   eax, DWORD PTR _rand_state[ecx], 1664525
    add    eax, 1013904223 ; 3c6ef35fH
    mov    DWORD PTR _rand_state[ecx], eax
    and    eax, 32767 ; 00007fffH
    ret    0
_my_rand ENDP

_TEXT    ENDS

```

rand\_state теперь в **TLS!**-сегменте и у каждого потока есть своя версия этой переменной.

Вот как к ней обращаться: загрузить адрес **TIB!** из FS:2Ch, затем прибавить дополнительный индекс (если нужно), затем вычислить адрес **TLS!**-сегмента.

Затем можно обращаться к переменной rand\_state через регистр ECX, который указывает на свою область в каждом потоке.

Селектор FS: знаком любому reverse engineer-у, он всегда указывает на **TIB!**, чтобы всегда можно было загружать данные специфичные для текущего потока.

В Win64 используется селектор GS: и адрес **TLS!** теперь 0x58:

Листинг 6.16: Оптимизирующий MSVC 2013 x64

```

_TLS    SEGMENT

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

rand_state DD    01H DUP (?)
_TLS      ENDS

_DATA    SEGMENT
$SG85451 DB      '%d', 0aH, 00H
_DATA    ENDS

_TEXT    SEGMENT

init$ = 8
my_srand PROC
    mov     edx, DWORD PTR _tls_index
    mov     rax, QWORD PTR gs:88 ; 58h
    mov     r8d, OFFSET FLAT:rand_state
    mov     rax, QWORD PTR [rax+rdx*8]
    mov     DWORD PTR [r8+rax], ecx
    ret     0
my_srand ENDP

my_rand PROC
    mov     rax, QWORD PTR gs:88 ; 58h
    mov     ecx, DWORD PTR _tls_index
    mov     edx, OFFSET FLAT:rand_state
    mov     rcx, QWORD PTR [rax+rcx*8]
    imul   eax, DWORD PTR [rcx+rdx], 1664525 ; 0019660dH
    add    eax, 1013904223 ; 3c6ef35fH
    mov     DWORD PTR [rcx+rdx], eax
    and    eax, 32767 ; 00007fffH
    ret     0
my_rand ENDP

_TEXT    ENDS

```

### Инициализированные данные в TLS!

Скажем, мы хотим, чтобы в переменной `rand_state` в самом начале было какое-то значение, и если программист забудет инициализировать генератор, то `rand_state` все же будет инициализирована какой-то константой (строка 9):

```

1 #include <stdint.h>
2 #include <windows.h>
3 #include <winnt.h>
4
5 // from the Numerical Recipes book:
6 #define RNG_a 1664525
7 #define RNG_c 1013904223
8
9 __declspec( thread ) uint32_t rand_state=1234;
10
11 void my_srand (uint32_t init)
12 {
13     rand_state=init;

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

14 }
15
16 int my_rand ()
17 {
18     rand_state=rand_state*RNG_a;
19     rand_state=rand_state+RNG_c;
20     return rand_state & 0x7fff;
21 }
22
23 int main()
24 {
25     printf ("%d\n", my_rand());
26 };

```

Код ничем не отличается от того, что мы уже видели, но вот что мы видим в IDA:

```

.tls:00404000 ; Segment type: Pure data
.tls:00404000 ; Segment permissions: Read/Write
.tls:00404000 _tls          segment para public 'DATA' use32
.tls:00404000             assume cs:_tls
.tls:00404000             ;org 404000h
.tls:00404000 TlsStart     db    0          ; DATA XREF:
                        .rdata:TlsDirectory
.tls:00404001             db    0
.tls:00404002             db    0
.tls:00404003             db    0
.tls:00404004             dd 1234
.tls:00404008 TlsEnd      db    0          ; DATA XREF: .rdata:TlsEnd_ptr
...

```

Там 1234 и теперь, во время запуска каждого нового потока, новый **TLS!** будет выделен для нового потока, и все эти данные, включая 1234, будут туда скопированы.

Вот типичный сценарий:

- Запустился поток А. **TLS!** создан для него, 1234 скопировалось в `rand_state`.
- Функция `my_rand()` была вызвана несколько раз в потоке А. `rand_state` теперь содержит что-то неравное 1234.
- Запустился поток Б. **TLS!** создан для него, 1234 скопировалось в `rand_state`, а в это же время, поток А имеет какое-то другое значение в этой переменной.

## TLS!-коллбэки

Но что если переменные в **TLS!** должны быть установлены в значения, которые должны быть подготовлены каким-то необычным образом?

Скажем, у нас есть следующая задача: программист может забыть вызвать функцию `my_srand()` для инициализации **PRNG!**, но генератор должен быть

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



инициализирован на старте чем-то по-настоящему случайным а не 1234.

Вот случай где можно применить **TLS!**-коллбэки.

Нижеследующий код не очень портабельный из-за хака, но тем не менее, вы поймете идею.

Мы здесь добавляем функцию (tls\_callback()), которая вызывается *перед* стартом процесса и/или потока.

Функция будет инициализировать **PRNG!** значением возвращенным функцией GetTickCount().

```
#include <stdint.h>
#include <windows.h>
#include <winnt.h>

// from the Numerical Recipes book:
#define RNG_a 1664525
#define RNG_c 1013904223

__declspec( thread ) uint32_t rand_state;

void my_srand (uint32_t init)
{
    rand_state=init;
}

void NTAPI tls_callback(PVOID a, DWORD dwReason, PVOID b)
{
    my_srand (GetTickCount());
}

#pragma data_seg(".CRT$XLB")
PIMAGE_TLS_CALLBACK p_thread_callback = tls_callback;
#pragma data_seg()

int my_rand ()
{
    rand_state=rand_state*RNG_a;
    rand_state=rand_state+RNG_c;
    return rand_state & 0x7fff;
}

int main()
{
    // rand_state is already initialized at the moment (using
    GetTickCount())
    printf ("%d\n", my_rand());
};
```

Посмотрим в IDA:

#### Листинг 6.17: Оптимизирующий MSVC 2013

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:00401020 TlsCallback_0 proc near ; DATA XREF:
                .rdata:TlsCallbacks
.text:00401020 call ds:GetTickCount
.text:00401026 push eax
.text:00401027 call my_srand
.text:0040102C pop ecx
.text:0040102D retn 0Ch
.text:0040102D TlsCallback_0 endp

...

.rdata:004020C0 TlsCallbacks dd offset TlsCallback_0 ; DATA XREF:
                .rdata:TlsCallbacks_ptr

...

.rdata:00402118 TlsDirectory dd offset TlsStart
.rdata:0040211C TlsEnd_ptr dd offset TlsEnd
.rdata:00402120 TlsIndex_ptr dd offset TlsIndex
.rdata:00402124 TlsCallbacks_ptr dd offset TlsCallbacks
.rdata:00402128 TlsSizeOfZeroFill dd 0
.rdata:0040212C TlsCharacteristics dd 300000h

```

TLS-коллбэки иногда используются в процедурах распаковки для запутывания их работы.

Некоторые люди могут быть в неведении что какой-то код уже был исполнен прямо перед **ОЕР!**<sup>4</sup>.

## Linux

Вот как глобальная переменная локальная для потока определяется в GCC:

```
__thread uint32_t rand_state=1234;
```

Этот модификатор не стандартный для Си/Си++, он присутствует только в GCC

<sup>5</sup>.

Селектор GS: также используется для доступа к **TLS!**, но немного иначе:

Листинг 6.18: Оптимизирующий GCC 4.8.1 x86

```

.text:08048460 my_srand proc near
.text:08048460
.text:08048460 arg_0 = dword ptr 4
.text:08048460
.text:08048460 mov eax, [esp+arg_0]
.text:08048464 mov gs:0FFFFFFFCh, eax
.text:0804846A retn
.text:0804846A my_srand endp

```

<sup>4</sup> **ОЕР!**

<sup>5</sup> <https://gcc.gnu.org/onlinedocs/gcc-3.3/gcc/C99-Thread-Local-Edits.html>

```

.text:08048470 my_rand      proc near
.text:08048470             imul   eax, gs:0FFFFFFFCh, 19660Dh
.text:0804847B             add    eax, 3C6EF35Fh
.text:08048480             mov    gs:0FFFFFFFCh, eax
.text:08048486             and    eax, 7FFFh
.text:0804848B             retn
.text:0804848B my_rand      endp

```

Еще об этом: [Ulrich Drepper, *ELF Handling For Thread-Local Storage*, (2013)]<sup>6</sup>.

## 6.3. Системные вызовы (syscall-ы)

Как известно, все работающие процессы в **OS!** делятся на две категории: имеющие полный доступ ко всему «железу» («kernel space») и не имеющие («user space»).

В первой категории ядро **OS!** и, обычно, драйвера.

Во второй категории всё прикладное ПО.

Например, ядро Linux в *kernel space*, но Glibc в *user space*.

Это разделение очень важно для безопасности **OS!**: очень важно чтобы никакой процесс не мог испортить что-то в других процессах или даже в самом ядре **OS!**. С другой стороны, падающий драйвер или ошибка внутри ядра **OS!** обычно приводит к kernel panic или **BSOD!**<sup>7</sup>.

Защита x86-процессора устроена так что возможно разделить всё на 4 слоя защиты (rings), но и в Linux, и в Windows, используются только 2: ring0 («kernel space») и ring3 («user space»).

Системные вызовы (syscall-ы) это точка где соединяются вместе оба эти пространства. Это, можно сказать, самое главное **API!** предоставляемое прикладному ПО.

В **Windows NT** таблица сисколлов находится в **SSDT!**<sup>8</sup>.

Работа через syscall-ы популярна у авторов шелл-кодов и вирусов, потому что там обычно бывает трудно определить адреса нужных функций в системных библиотеках, а syscall-ами проще пользоваться, хотя и придется писать больше кода из-за более низкого уровня абстракции этого **API!**. Также нельзя еще забывать, что номера syscall-ов могут отличаться от версии к версии OS.

### 6.3.1. Linux

В Linux вызов syscall-а обычно происходит через `int 0x80`. В регистре EAX передается номер вызова, в остальных регистрах — параметры.

<sup>6</sup>Также доступно здесь: <http://www.akkadia.org/drepper/tls.pdf>

<sup>7</sup>**BSOD!**

<sup>8</sup>**SSDT!**

Листинг 6.19: Простой пример использования пары syscall-ов

```

section .text
global _start

_start:
    mov     edx,len ; buffer len
    mov     ecx,msg ; buffer
    mov     ebx,1   ; file descriptor. 1 is for stdout
    mov     eax,4   ; syscall number. 4 is for sys_write
    int     0x80

    mov     eax,1   ; syscall number. 1 is for sys_exit
    int     0x80

section .data
msg     db  'Hello, world!',0xa
len     equ $ - msg

```

Компиляция:

```

nasm -f elf32 1.s
ld 1.o

```

Полный список syscall-ов в Linux: <http://syscalls.kernelgrok.com/>.

Для перехвата и трассировки системных вызовов в Linux, можно применять `strace(?? (стр. ??))`.

### 6.3.2. Windows

Вызов происходит через `int 0x2e` либо используя специальную x86-инструкцию `SYSENTER`.

Полный список syscall-ов в Windows: <http://j00ru.vexillum.org/ntapi/>.

Смотрите также:

«Windows Syscall Shellcode» by Piotr Bania: <http://www.symantec.com/connect/articles/windows-syscall-shellcode>.

## 6.4. Linux

### 6.4.1. Адресно-независимый код

Во время анализа динамических библиотек (.so) в Linux, часто можно заметить такой шаблонный код:

Листинг 6.20: libc-2.17.so x86

```

.text:0012D5E3 __x86_get_pc_thunk_bx proc near ; CODE XREF: sub_17350+3
.text:0012D5E3 ; sub_173CC+4 ...

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:0012D5E3     mov     ebx, [esp+0]
.text:0012D5E6     retn
.text:0012D5E6     __x86_get_pc_thunk_bx endp

...

.text:000576C0 sub_576C0      proc near      ; CODE XREF: tmpfile+73
...

.text:000576C0     push   ebp
.text:000576C1     mov    ecx, large gs:0
.text:000576C8     push   edi
.text:000576C9     push   esi
.text:000576CA     push   ebx
.text:000576CB     call  __x86_get_pc_thunk_bx
.text:000576D0     add    ebx, 157930h
.text:000576D6     sub    esp, 9Ch

...

.text:000579F0     lea    eax, (a__gen_tempname - 1AF000h)[ebx] ;
                "__gen_tempname"
.text:000579F6     mov    [esp+0ACh+var_A0], eax
.text:000579FA     lea    eax, (a__SysdepsPosix - 1AF000h)[ebx] ;
                "../sysdeps/posix/tempname.c"
.text:00057A00     mov    [esp+0ACh+var_A8], eax
.text:00057A04     lea    eax, (aInvalidKindIn_ - 1AF000h)[ebx] ;
                "! \"invalid KIND in __gen_tempname\""
.text:00057A0A     mov    [esp+0ACh+var_A4], 14Ah
.text:00057A12     mov    [esp+0ACh+var_AC], eax
.text:00057A15     call  __assert_fail

```

Все указатели на строки корректируются при помощи некоторой константы из регистра EBX, которая вычисляется в начале каждой функции.

Это так называемый адресно-независимый код (**PIC!**), он предназначен для исполнения будучи расположенным по любому адресу в памяти, вот почему он не содержит никаких абсолютных адресов в памяти.

**PIC!** был очень важен в ранних компьютерных системах и важен сейчас во встраиваемых<sup>9</sup>, не имеющих поддержки виртуальной памяти (все процессы расположены в одном непрерывном блоке памяти). Он до сих пор используется в \*NIX системах для динамических библиотек, потому что динамическая библиотека может использоваться одновременно в нескольких процессах, будучи загружена в память только один раз. Но все эти процессы могут загрузить одну и ту же динамическую библиотеку по разным адресам, вот почему динамическая библиотека должна работать корректно, не привязываясь к абсолютным адресам.

Простой эксперимент:

---

<sup>9</sup>embedded

```
#include <stdio.h>

int global_variable=123;

int f1(int var)
{
    int rt=global_variable+var;
    printf ("returning %d\n", rt);
    return rt;
};
```

Скомпилируем в GCC 4.7.3 и посмотрим итоговый файл .so в **IDA!**:

```
gcc -fPIC -shared -O3 -o l.so l.c
```

#### Листинг 6.21: GCC 4.7.3

```
.text:00000440      public __x86_get_pc_thunk_bx
.text:00000440      __x86_get_pc_thunk_bx proc near          ;
                    CODE XREF: _init_proc+4
.text:00000440      ;
                    deregister_tm_clones+4 ...
.text:00000440      mov     ebx, [esp+0]
.text:00000443      retn
.text:00000443      __x86_get_pc_thunk_bx endp

.text:00000570      public f1
.text:00000570      f1     proc near
.text:00000570
.text:00000570      var_1C = dword ptr -1Ch
.text:00000570      var_18 = dword ptr -18h
.text:00000570      var_14 = dword ptr -14h
.text:00000570      var_8  = dword ptr -8
.text:00000570      var_4  = dword ptr -4
.text:00000570      arg_0  = dword ptr 4
.text:00000570
.text:00000570      sub     esp, 1Ch
.text:00000573      mov     [esp+1Ch+var_8], ebx
.text:00000577      call   __x86_get_pc_thunk_bx
.text:0000057C      add     ebx, 1A84h
.text:00000582      mov     [esp+1Ch+var_4], esi
.text:00000586      mov     eax, ds:(global_variable_ptr - 2000h)
                    ↙ ) [ebx]
.text:0000058C      mov     esi, [eax]
.text:0000058E      lea    eax, (aReturningD - 2000h)[ebx] ;
                    "returning %d\n"
.text:00000594      add     esi, [esp+1Ch+arg_0]
.text:00000598      mov     [esp+1Ch+var_18], eax
.text:0000059C      mov     [esp+1Ch+var_1C], 1
.text:000005A3      mov     [esp+1Ch+var_14], esi
.text:000005A7      call   __printf_chk
.text:000005AC      mov     eax, esi
.text:000005AE      mov     ebx, [esp+1Ch+var_8]
.text:000005B2      mov     esi, [esp+1Ch+var_4]
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мой адреса](#). Спасибо!

```
.text:000005B6      add     esp, 1Ch
.text:000005B9      retn
.text:000005B9 f1      endp
```

Так и есть: указатели на строку «*returning %d\n*» и переменную *global\_variable* корректируются при каждом исполнении функции.

Функция `__x86_get_pc_thunk_bx()` возвращает адрес точки после вызова самой себя (здесь: `0x57C`) в `EBX`. Это очень простой способ получить значение указателя на текущую инструкцию (EIP) в произвольном месте.

Константа `0x1A84` связана с разницей между началом этой функции и так называемой *Global Offset Table Procedure Linkage Table* (GOT PLT), секцией, сразу же за *Global Offset Table* (GOT), где находится указатель на *global\_variable*. **IDA!** показывает смещения уже обработанными, чтобы их было проще понимать, но на самом деле код такой:

```
.text:00000577      call   __x86_get_pc_thunk_bx
.text:0000057C      add     ebx, 1A84h
.text:00000582      mov     [esp+1Ch+var_4], esi
.text:00000586      mov     eax, [ebx-0Ch]
.text:0000058C      mov     esi, [eax]
.text:0000058E      lea    eax, [ebx-1A30h]
```

Так что, `EBX` указывает на секцию GOT PLT и для вычисления указателя на *global\_variable*, которая хранится в GOT, нужно вычесть `0xC`. А чтобы вычислить указатель на «*returning %d\n*», нужно вычесть `0x1A30`.

Кстати, вот зачем в AMD64 появилась поддержка адресации относительно `RIP`<sup>10</sup>, просто для упрощения PIC-кода.

Скомпилируем тот же код на Си при помощи той же версии GCC, но для x64.

**IDA!** упростит код на выходе убирая упоминания `RIP`, так что будем использовать *objdump* вместо нее:

```
00000000000000720 <f1>:
720:  48 8b 05 b9 08 20 00      mov     rax,QWORD PTR [rip+0x2008b9]
    ;
    200fe0 <_DYNAMIC+0x1d0>
727:  53                       push   rbx
728:  89 fb                     mov     ebx,edi
72a:  48 8d 35 20 00 00 00      lea    rsi,[rip+0x20]
    751 <_fini+0x9>
731:  bf 01 00 00 00           mov     edi,0x1
736:  03 18                     add     ebx,DWORD PTR [rax]
738:  31 c0                     xor     eax,eax
73a:  89 da                     mov     edx,ebx
73c:  e8 df fe ff ff          call   620 <__printf_chk@plt>
741:  89 d8                     mov     eax,ebx
743:  5b                       pop    rbx
744:  c3                       ret
```

<sup>10</sup>указатель инструкций в AMD64

0x2008b9 это разница между адресом инструкции по 0x720 и *global\_variable*, а 0x20 это разница между инструкцией по 0x72A и строкой «*returning %d\n*».

Как видно, необходимость очень часто пересчитывать адреса делает исполнение немного медленнее (хотя это и стало лучше в x64). Так что если вы заботитесь о скорости исполнения, то, наверное, нужно задуматься о статической компоновке (static linking) [см. Agner Fog, *Optimizing software in C++* (2015)].

## Windows

Такой механизм не используется в Windows DLL. Если загрузчику в Windows приходится загружать DLL в другое место, он «патчит» DLL прямо в памяти (на местах *FIXUP*-ов) чтобы скорректировать все адреса. Это приводит к тому что загруженную один раз DLL нельзя использовать одновременно в разных процессах, желающих расположить её по разным адресам — потому что каждый загруженный в память экземпляр DLL *доводится* до того чтобы работать только по этим адресам.

### 6.4.2. Трюк с *LD\_PRELOAD* в Linux

Это позволяет загружать свои динамические библиотеки перед другими, даже перед системными, такими как *libc.so.6*.

Что в свою очередь, позволяет «подставлять» написанные нами функции перед оригинальными из системных библиотек.

Например, легко перехватывать все вызовы к *time()*, *read()*, *write()*, и т. д.

Попробуем узнать, сможем ли мы обмануть утилиту *uptime*. Как известно, она сообщает, как долго компьютер работает. При помощи *strace(??* (стр. ??)), можно увидеть, что эту информацию утилита получает из файла */proc/uptime*:

```
$ strace uptime
...
open("/proc/uptime", O_RDONLY)      = 3
lseek(3, 0, SEEK_SET)              = 0
read(3, "416166.86 414629.38\n", 2047) = 20
...
```

Это не реальный файл на диске, это виртуальный файл, содержимое которого генерируется на лету в ядре Linux.

Там просто два числа:

```
$ cat /proc/uptime
416690.91 415152.03
```

Из Wikipedia, можно узнать <sup>11</sup>:

<sup>11</sup><https://en.wikipedia.org/wiki/Uptime>



The first number is the total number of seconds the system has been up. The second number is how much of that time the machine has spent idle, in seconds.

Попробуем написать свою динамическую библиотеку, в которой будет `open()`, `read()`, `close()` с нужной нам функциональностью.

Во-первых, наш `open()` будет сравнивать имя открываемого файла с тем что нам нужно, и если да, то будет запоминать дескриптор открытого файла.

Во-вторых, `read()`, если будет вызываться для этого дескриптора, будет подменять вывод, а в остальных случаях, будет вызывать настоящий `read()` из `libc.so.6`. А также `close()`, будет следить, закрывается ли файл за которым мы следим.

Для того чтобы найти адреса настоящих функций в `libc.so.6`, используем `dlopen()` и `dlsym()`.

Нам это нужно, потому что нам нужно передавать управление «настоящим» функциями.

С другой стороны, если бы мы перехватывали, скажем, `strcmp()`, и следили бы за всеми сравнениями строк в программе, то, наверное, `strcmp()` можно было бы и самому реализовать, не пользуясь настоящей функцией <sup>12</sup>, так было бы проще.

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <dlfcn.h>
#include <string.h>

void *libc_handle = NULL;
int (*open_ptr)(const char *, int) = NULL;
int (*close_ptr)(int) = NULL;
ssize_t (*read_ptr)(int, void*, size_t) = NULL;

bool initied = false;

_Noreturn void die (const char * fmt, ...)
{
    va_list va;
    va_start (va, fmt);

    vprintf (fmt, va);
    exit(0);
};
```

<sup>12</sup>Например, посмотрите как обеспечивается простейший перехват `strcmp()` в статье <sup>13</sup> написанной Yong Huang

```
static void find_original_functions ()
{
    if (inited)
        return;

    libc_handle = dlopen ("libc.so.6", RTLD_LAZY);
    if (libc_handle==NULL)
        die ("can't open libc.so.6\n");

    open_ptr = dlsym (libc_handle, "open");
    if (open_ptr==NULL)
        die ("can't find open()\n");

    close_ptr = dlsym (libc_handle, "close");
    if (close_ptr==NULL)
        die ("can't find close()\n");

    read_ptr = dlsym (libc_handle, "read");
    if (read_ptr==NULL)
        die ("can't find read()\n");

    inited = true;
}

static int opened_fd=0;

int open(const char *pathname, int flags)
{
    find_original_functions();

    int fd>(*open_ptr)(pathname, flags);
    if (strcmp(pathname, "/proc/uptime")==0)
        opened_fd=fd; // that's our file! record its file descriptor
    else
        opened_fd=0;
    return fd;
};

int close(int fd)
{
    find_original_functions();

    if (fd==opened_fd)
        opened_fd=0; // the file is not opened anymore
    return (*close_ptr)(fd);
};

ssize_t read(int fd, void *buf, size_t count)
{
    find_original_functions();

    if (opened_fd!=0 && fd==opened_fd)
    {
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        // that's our file!
        return sprintf (buf, count, "%d %d", 0x7fffffff, 0x
↳ x7fffffff)+1;
    };
    // not our file, go to real read() function
    return (*read_ptr)(fd, buf, count);
};

```

( [Исходный код](#) )

Компилируем как динамическую библиотеку:

```
gcc -fpic -shared -Wall -o fool_uptime.so fool_uptime.c -ldl
```

Запускаем *uptime*, подгружая нашу библиотеку перед остальными:

```
LD_PRELOAD=`pwd`/fool_uptime.so uptime
```

Видим такое:

```
01:23:02 up 24855 days,  3:14,  3 users,  load average: 0.00, 0.01, 0.05
```

Если переменная окружения *LD\_PRELOAD* будет всегда указывать на путь и имя файла нашей библиотеки, то она будет загружаться для всех запускаемых программ.

Еще примеры:

- Перехват `time()` в Sun Solaris [yurichev.com](http://yurichev.com)
- Очень простой перехват `strcmp()` (Yong Huang) [https://yurichev.com/mirrors/LD\\_PRELOAD/Yong%20Huang%20LD\\_PRELOAD.txt](https://yurichev.com/mirrors/LD_PRELOAD/Yong%20Huang%20LD_PRELOAD.txt)
- Kevin Pulo — Fun with `LD_PRELOAD`. Много примеров и идей. [yurichev.com](http://yurichev.com)
- Перехват функций работы с файлами для компрессии и декомпрессии файлов на лету (zlibc). <ftp://metalab.unc.edu/pub/Linux/libs/compression>

## 6.5. Windows NT

### 6.5.1. CRT (win32)

Начинается ли исполнение программы прямо с функции `main()`? Нет, не начинается. Если открыть любой исполняемый файл в **IDA!** или **Niew**, то **OEP!** указывает на какой-то совсем другой код.

Это код, который делает некоторые приготовления перед тем как запустить ваш код. Он называется стартап-код или CRT-код (C RunTime).

Функция `main()` принимает на вход массив из параметров, переданных в командной строке, а также переменные окружения. Но в реальности в программу передается командная строка в виде простой строки, это именно CRT-код

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

находит там пробелы и разрезает строку на части. CRT-код также готовит массив переменных окружения envp. В **GUI!**<sup>14</sup>-приложениях win32, вместо main() имеется функция WinMain со своими аргументами:

```
int CALLBACK WinMain(
    _In_ HINSTANCE hInstance,
    _In_ HINSTANCE hPrevInstance,
    _In_ LPSTR lpCmdLine,
    _In_ int nCmdShow
);
```

CRT-код готовит и их.

А также, число, возвращаемое функцией main(), это код ошибки возвращаемый программой. В CRT это значение передается в ExitProcess(), принимающей в качестве аргумента код ошибки.

Как правило, каждый компилятор имеет свой CRT-код.

Вот типичный для MSVC 2008 CRT-код.

```
1  __tmainCRTStartup proc near
2
3  var_24 = dword ptr -24h
4  var_20 = dword ptr -20h
5  var_1C = dword ptr -1Ch
6  ms_exc = CPPEH_RECORD ptr -18h
7
8      push    14h
9      push    offset stru_4092D0
10     call    __SEH_prolog4
11     mov     eax, 5A4Dh
12     cmp     ds:400000h, ax
13     jnz     short loc_401096
14     mov     eax, ds:40003Ch
15     cmp     dword ptr [eax+400000h], 4550h
16     jnz     short loc_401096
17     mov     ecx, 10Bh
18     cmp     [eax+400018h], cx
19     jnz     short loc_401096
20     cmp     dword ptr [eax+400074h], 0Eh
21     jbe     short loc_401096
22     xor     ecx, ecx
23     cmp     [eax+4000E8h], ecx
24     setnz  cl
25     mov     [ebp+var_1C], ecx
26     jmp     short loc_40109A
27
28
29 loc_401096: ; CODE XREF: __tmainCRTStartup+18
30             ; __tmainCRTStartup+29 ...
31     and     [ebp+var_1C], 0
```

<sup>14</sup>**GUI!**

```
32
33 loc_40109A: ; CODE XREF: ___tmainCRTStartup+50
34     push    1
35     call   __heap_init
36     pop    ecx
37     test   eax, eax
38     jnz   short loc_4010AE
39     push  1Ch
40     call  _fast_error_exit
41     pop   ecx
42
43 loc_4010AE: ; CODE XREF: ___tmainCRTStartup+60
44     call  __mtinit
45     test  eax, eax
46     jnz  short loc_4010BF
47     push  10h
48     call  _fast_error_exit
49     pop   ecx
50
51 loc_4010BF: ; CODE XREF: ___tmainCRTStartup+71
52     call  sub_401F2B
53     and   [ebp+ms_exc.disabled], 0
54     call  __ioinit
55     test  eax, eax
56     jge  short loc_4010D9
57     push  1Bh
58     call  __amsg_exit
59     pop   ecx
60
61 loc_4010D9: ; CODE XREF: ___tmainCRTStartup+8B
62     call  ds:GetCommandLineA
63     mov   dword_40B7F8, eax
64     call  ___crtGetEnvironmentStringsA
65     mov   dword_40AC60, eax
66     call  __setargv
67     test  eax, eax
68     jge  short loc_4010FF
69     push  8
70     call  __amsg_exit
71     pop   ecx
72
73 loc_4010FF: ; CODE XREF: ___tmainCRTStartup+B1
74     call  __setenvp
75     test  eax, eax
76     jge  short loc_401110
77     push  9
78     call  __amsg_exit
79     pop   ecx
80
81 loc_401110: ; CODE XREF: ___tmainCRTStartup+C2
82     push  1
83     call  __cinit
84     pop   ecx
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

85     test    eax, eax
86     jz      short loc_401123
87     push   eax
88     call   __amsg_exit
89     pop    ecx
90
91 loc_401123: ; CODE XREF: ___tmainCRTStartup+D6
92     mov    eax, envp
93     mov    dword_40AC80, eax
94     push   eax           ; envp
95     push   argv          ; argv
96     push   argc          ; argc
97     call   _main
98     add    esp, 0Ch
99     mov    [ebp+var_20], eax
100    cmp    [ebp+var_1C], 0
101    jnz    short $LN28
102    push   eax           ; uExitCode
103    call   $LN32
104
105 $LN28:     ; CODE XREF: ___tmainCRTStartup+105
106     call   __cexit
107     jmp    short loc_401186
108
109
110 $LN27:     ; DATA XREF: .rdata:stru_4092D0
111     mov    eax, [ebp+ms_exc.exc_ptr] ; Exception filter 0 for function
112     401044
113     mov    ecx, [eax]
114     mov    ecx, [ecx]
115     mov    [ebp+var_24], ecx
116     push   eax
117     push   ecx
118     call   __XcptFilter
119     pop    ecx
120     pop    ecx
121
122 $LN24:
123     retn
124
125 $LN14:     ; DATA XREF: .rdata:stru_4092D0
126     401044
127     mov    esp, [ebp+ms_exc.old_esp] ; Exception handler 0 for function
128     mov    eax, [ebp+var_24]
129     mov    [ebp+var_20], eax
130     cmp    [ebp+var_1C], 0
131     jnz    short $LN29
132     push   eax           ; int
133     call   __exit
134
135 $LN29:     ; CODE XREF: ___tmainCRTStartup+135
136     call   __c_exit

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

137
138 loc_401186: ; CODE XREF: __tmainCRTStartup+112
139     mov     [ebp+ms_exc.disabled], 0FFFFFFEh
140     mov     eax, [ebp+var_20]
141     call   __SEH_epilog4
142     retn

```

Здесь можно увидеть по крайней мере вызов функции `GetCommandLineA()` (строка 62), затем `setargv()` (строка 66) и `setenvp()` (строка 74), которые, видимо, заполняют глобальные переменные-указатели `argc`, `argv`, `envp`.

В итоге, вызывается `main()` с этими аргументами (строка 97).

Также имеются вызовы функций с говорящими именами вроде `heap_init()` (строка 35), `ioinit()` (строка 54).

**Куча** действительно инициализируется в **CRT!**. Если вы попытаетесь использовать `malloc()` в программе без CRT, программа упадет с такой ошибкой:

```

runtime error R6030
- CRT not initialized

```

Инициализация глобальных объектов в Си++ происходит до вызова `main()`, именно в **CRT!**: ?? (стр. ??).

Значение, возвращаемое из `main()` передается или в `sexit()`, или же в `$LN32`, которая далее вызывает `doexit()`.

Можно ли обойтись без **CRT!**? Можно, если вы знаете что делаете.

В линкере от **MSVC!** точка входа задается опцией `/ENTRY`.

```

#include <windows.h>

int main()
{
    MessageBox (NULL, "hello, world", "caption", MB_OK);
};

```

Компилируем в MSVC 2008.

```

cl no_crt.c user32.lib /link /entry:main

```

Получаем вполне работающий `.exe` размером 2560 байт, внутри которого есть только PE-заголовок, инструкции, вызывающие `MessageBox`, две строки в сегменте данных, импортируемая из `user32.dll` функция `MessageBox`, и более ничего.

Это работает, но вы уже не сможете вместо `main()` написать `WinMain` с его четырьмя аргументами. Вернее, если быть точным, написать-то сможете, но доступа к этим аргументам не будет, потому что они не подготовлены на момент исполнения.

Кстати, можно еще короче сделать `.exe` если уменьшить выравнивание **PE!**-секций (которое, по умолчанию, 4096 байт).

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
cl no_crt.c user32.lib /link /entry:main /align:16
```

Линкер скажет:

```
LINK : warning LNK4108: /ALIGN specified without /DRIVER; image may not run
```

Получим .exe размером 720 байт. Он запускается в Windows 7 x86, но не x64 (там выдает ошибку при загрузке). При желании, размер можно еще сильнее ужать, но, как видно, возникают проблемы с совместимостью с разными версиями Windows.

## 6.5.2. Win32 PE

**PE!** это формат исполняемых файлов, принятый в Windows.

Разница между .exe, .dll, и .sys в том, что у .exe и .sys обычно нет экспортов, только импорты.

У **DLL!**<sup>15</sup>, как и у всех PE-файлов, есть точка входа (**OEP!**) (там располагается функция DllMain()), но обычно эта функция ничего не делает.

.sys это обычно драйвера устройств.

Для драйверов, Windows требует, чтобы контрольная сумма в PE-файле была проставлена и была верной<sup>16</sup>.

А начиная с Windows Vista, файлы драйверов должны быть также подписаны при помощи электронной подписи, иначе они не будут загружаться.

В начале всякого PE-файла есть крохотная DOS-программа, выводящая на консоль сообщение вроде «This program cannot be run in DOS mode.» — если запустить эту программу в DOS либо Windows 3.1 (**OS!** не знающие о PE-формате), выведется это сообщение.

### Терминология

- Модуль — это отдельный файл, .exe или .dll.
- Процесс — это некая загруженная в память и работающая программа. Как правило, состоит из одного .exe-файла и массы .dll-файлов.
- Память процесса — память с которой работает процесс. У каждого процесса — своя. Там обычно имеются загруженные модули, память стека, [кучи](#), и т. д.
- **VA!**<sup>17</sup> — это адрес, который будет использоваться в самой программе во время исполнения.

---

<sup>15</sup> **DLL!**

<sup>16</sup> Например, Niew(??) умеет её подсчитывать

<sup>17</sup> **VA!**



- Базовый адрес (модуля) — это адрес, по которому модуль должен быть загружен в пространство процесса.  
Загрузчик **OS!** может его изменить, если этот базовый адрес уже занят другим модулем, загруженным перед ним.
- **RVA!**<sup>18</sup> — это **VA!**-адрес минус базовый адрес. Многие адреса в таблицах PE-файла используют **RVA!**-адреса.
- **IAT!**<sup>19</sup> — массив адресов импортированных символов<sup>20</sup>.  
Иногда, директория `IMAGE_DIRECTORY_ENTRY_IAT` указывает на **IAT!**. Важно отметить, что **IDA!** (по крайней мере 6.1) может выделить псевдо-секцию с именем `.idata` для **IAT!**, даже если **IAT!** является частью совсем другой секции!
- **INT!**<sup>21</sup> — массив имен символов для импортирования<sup>22</sup>.

### Базовый адрес

Дело в том, что несколько авторов модулей могут готовить DLL-файлы для других, и нет возможности договориться о том, какие адреса и кому будут отведены.

Поэтому, если у двух необходимых для загрузки процесса DLL одинаковые базовые адреса, одна из них будет загружена по этому базовому адресу, а вторая — по другому свободному месту в памяти процесса, и все виртуальные адреса во второй DLL будут скорректированы.

Очень часто линкер в **MSVC!** генерирует `.exe`-файлы с базовым адресом `0x400000`<sup>23</sup>, и с секцией кода начинающейся с `0x401000`. Это значит, что **RVA!** начала секции кода — `0x1000`. А **DLL!** часто генерируются **MSVC**-линкером с базовым адресом `0x10000000`<sup>24</sup>.

Помимо всего прочего, есть еще одна причина намеренно загружать модули по разным адресам, а точнее, по случайным.

Это **ASLR!**.

Дело в том, что некий шелл-код, пытающийся исполниться на зараженной системе, должен вызывать какие-то системные функции, а следовательно, знать их адреса.

И в старых **OS!** (в линейке **Windows NT**: до **Windows Vista**), системные DLL (такие как `kernel32.dll`, `user32.dll`) загружались все время по одним и тем же адресам, а если еще и вспомнить, что версии этих DLL редко менялись, то адреса отдельных функций, можно сказать, фиксированы и шелл-код может вызывать их напрямую.

<sup>18</sup> **RVA!**

<sup>19</sup> **IAT!**

<sup>20</sup> Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)]

<sup>21</sup> **INT!**

<sup>22</sup> Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)]

<sup>23</sup> Причина выбора такого адреса описана здесь: [MSDN](#)

<sup>24</sup> Это можно изменять опцией `/BASE` в линкере

Чтобы избежать этого, методика **ASLR!** загружает и вашу программу, и все модули ей необходимые, по случайным адресам, разным при каждом запуске.

В PE-файлах, поддержка **ASLR!** отмечается выставлением флага `IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE` [см: Mark Russinovich, *Microsoft Windows Internals*].

## Subsystem

Имеется также поле *subsystem*, обычно это:

- native<sup>25</sup> (.sys-драйвер),
- console (консольное приложение) или
- **GUI!** (не консольное).

## Версия ОС

В PE-файле также задается минимальный номер версии Windows, необходимый для загрузки модуля.

Соответствие номеров версий в файле и кодовых наименований Windows, можно посмотреть здесь<sup>26</sup>.

Например, **MSVC!** 2005 еще компилирует .exe-файлы запускающиеся на Windows NT4 (версия 4.00), а вот **MSVC!** 2008 уже нет (генерируемые файлы имеют версию 5.00, для запуска необходима как минимум Windows 2000).

**MSVC!** 2012 по умолчанию генерирует .exe-файлы версии 6.00, для запуска нужна как минимум Windows Vista. Хотя, изменив настройки компиляции<sup>27</sup>, можно заставить генерировать и под Windows XP.

## Секции

Разделение на секции присутствует, по-видимому, во всех форматах исполняемых файлов.

Придумано это для того, чтобы отделить код от данных, а данные — от константных данных.

- На секции кода будет стоять флаг `IMAGE_SCN_CNT_CODE` или `IMAGE_SCN_MEM_EXECUTE` — это исполняемый код.
- На секции данных — флаги `IMAGE_SCN_CNT_INITIALIZED_DATA`, `IMAGE_SCN_MEM_READ` и `IMAGE_SCN_MEM_WRITE`.
- На пустой секции с неинициализированными данными — `IMAGE_SCN_CNT_UNINITIALIZED_DATA`, `IMAGE_SCN_MEM_READ` и `IMAGE_SCN_MEM_WRITE`.

<sup>25</sup>Что означает, что модуль использует Native API а не Win32

<sup>26</sup>[Wikipédia](#)

<sup>27</sup>[MSDN](#)

- А на секции с константными данными, то есть, защищенными от записи, могут быть флаги `IMAGE_SCN_CNT_INITIALIZED_DATA` и `IMAGE_SCN_MEM_READ` без `IMAGE_SCN_MEM_WRITE`. Если попытаться записать что-то в эту секцию, процесс упадет.

В PE-файле можно задавать название для секции, но это не важно. Часто (но не всегда) секция кода называется `.text`, секция данных — `.data`, константных данных — `.rdata` (*readable data*) (возможно, имеется ввиду также *read-only-data*). Еще популярные имена секций:

- `.idata` — секция импортов. **IDA!** может создавать псевдо-секцию с этим же именем: ?? (стр. ??).
- `.edata` — секция экспортов (редко встречается)
- `.pdata` — секция содержащая информацию об исключениях в Windows NT для MIPS, **IA64!** и x64: ?? (стр. ??)
- `.reloc` — секция релоков
- `.bss` — неинициализированные данные
- `.tls` — thread local storage (**TLS!**)
- `.rsrc` — ресурсы
- `.CRT` — может присутствовать в бинарных файлах, скомпилированных очень старыми версиями MSVC

Запаковщики/зашифровщики PE-файлов часто затирают имена секций, или меняют на свои.

В **MSVC!** можно объявлять данные в произвольно названной секции<sup>28</sup>.

Некоторые компиляторы и линкеры могут добавлять также секцию с отладочными символами и вообще отладочной информацией (например, MinGW).

Хотя это не так в современных версиях **MSVC!** (там принято отладочную информацию сохранять в отдельных **PDB**-файлах).

Вот как PE-секция описывается в файле:

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD PhysicalAddress;
        DWORD VirtualSize;
    } Misc;
    DWORD VirtualAddress;
    DWORD SizeOfRawData;
    DWORD PointerToRawData;
    DWORD PointerToRelocations;
    DWORD PointerToLinenumbers;
    WORD  NumberOfRelocations;
    WORD  NumberOfLinenumbers;
    DWORD Characteristics;
}
```

<sup>28</sup>[MSDN](#)

```
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

29

Еще немного терминологии: *PointerToRawData* называется «Offset» в Hiew и *VirtualAddress* называется «RVA» там же.

### Секция данных

Секция данных в файле может быть меньше, чем в памяти. Например, некоторые переменные могут быть инициализированы, а некоторые — нет. Тогда компилятор и линкер объединяют их все в одну секцию, но первая часть секции инициализирована и находится в файле, а вторая отсутствует в файле (конечно, для экономии места). *VirtualSize* будет равен размеру секции в памяти, а *SizeOfRawData* будет равен размеру секции в файле.

IDA будет показывать границу между инициализированной и неинициализированной частями так:

```
...
.data:10017FFA          db      0
.data:10017FFB          db      0
.data:10017FFC          db      0
.data:10017FFD          db      0
.data:10017FFE          db      0
.data:10017FFF          db      0
.data:10018000          db      ? ;
.data:10018001          db      ? ;
.data:10018002          db      ? ;
.data:10018003          db      ? ;
.data:10018004          db      ? ;
.data:10018005          db      ? ;
...
```

### .rdata — секция данных только для чтения

Тут обыкновенно располагаются строки (потому что они имеют тип `const char*`, другие переменные отмеченные как *const*, имена импортируемых ф-ций.

См. также: ?? (стр. ??).

### Релоки

Также известны как FIXUP-ы (по крайней мере в Hiew).

Это также присутствует почти во всех форматах загружаемых и исполняемых файлов <sup>30</sup>.

<sup>29</sup>MSDN

<sup>30</sup>Даже .exe-файлы в MS-DOS

Исключения это динамические библиотеки явно скомпилированные с **PIC!** или любой другой **PIC!**-код.

Зачем они нужны? Как видно, модули могут загружаться по другим базовым адресам, но как же тогда работать с глобальными переменными, например?

Ведь нужно обращаться к ним по адресу. Одно из решений — это адресно-независимый код (**??** (стр. **??**)). Но это далеко не всегда удобно.

Поэтому имеется таблица релоков. Там просто перечислены адреса мест в модуле подлежащими коррекции при загрузке по другому базовому адресу.

Например, по 0x410000 лежит некая глобальная переменная, и вот как обеспечивается её чтение:

A1 00 00 41 00	mov	eax, [000410000]
----------------	-----	------------------

Базовый адрес модуля 0x400000, а **RVA!** глобальной переменной 0x10000.

Если загружать модуль по базовому адресу 0x500000, нужно чтобы адрес этой переменной в этой инструкции стал 0x510000.

Как видно, адрес переменной закодирован в самой инструкции MOV, после байта 0xA1.

Поэтому адрес четырех байт, после 0xA1, записывается в таблицу релоков.

Если модуль загружается по другому базовому адресу, загрузчик **OS!** обходит все адреса в таблице, находит каждое 32-битное слово по этому адресу, отнимает от него настоящий, оригинальный базовый адрес (в итоге получается **RVA!**), и прибавляет к нему новый базовый адрес.

А если модуль загружается по своему оригинальному базовому адресу, ничего не происходит.

Так можно обходиться со всеми глобальными переменными.

Релоки могут быть разных типов, однако в Windows для x86-процессоров, тип обычно *IMAGE\_REL\_BASED\_HIGHLOW*.

Кстати, релоки маркируются темным в Niew, например: илл. **??**. (Эти места нужно обходить в процессе патчинга.)

OllyDbg подчеркивает места в памяти, к которым будут применены релоки, например: илл. **??**.

## Экспорты и импорты

Как известно, любая исполняемая программа должна как-то пользоваться сервисами **OS!** и прочими DLL-библиотеками.

Можно сказать, что нужно связывать функции из одного модуля (обычно DLL) и места их вызовов в другом модуле (.exe-файл или другая DLL).

Для этого, у каждой DLL есть «экспорты», это таблица функций плюс их адреса в модуле.

А у .exe-файла, либо DLL, есть «импорты», это таблица функций требующихся для исполнения включая список имен DLL-файлов.

Загрузчик **OS!**, после загрузки основного .exe-файла, проходит по таблице импортов: загружает дополнительные DLL-файлы, находит имена функций среди экспортов в DLL и прописывает их адреса в **IAT!** в головном .exe-модуле.

Как видно, во время загрузки, загрузчику нужно много сравнивать одни имена функций с другими, а сравнение строк — это не очень быстрая процедура, так что, имеется также поддержка «ординалов» или «hint»-ов, это когда в таблице импортов проставлены номера функций вместо их имен.

Так их быстрее находить в загружаемой DLL. В таблице экспортов ординалы присутствуют всегда.

К примеру, программы использующие библиотеки **MFC!**<sup>31</sup>, обычно загружают mfc\*.dll по ординалам, и в таких программах, в **INT!**, нет имен функций **MFC!**.

При загрузке такой программы в **IDA!**, она спросит у вас путь к файлу mfc\*.dll, чтобы установить имена функций. Если в **IDA!** не указать путь к этой DLL, то вместо имен функций будет что-то вроде *mfc80\_123*.

### Секция импортов

Под таблицу импортов и всё что с ней связано иногда отводится отдельная секция (с названием вроде .idata), но это не обязательно.

Импорты — это запутанная тема еще и из-за терминологической путаницы. Попробуем собрать всё в одно место.

---

<sup>31</sup>**MFC!**

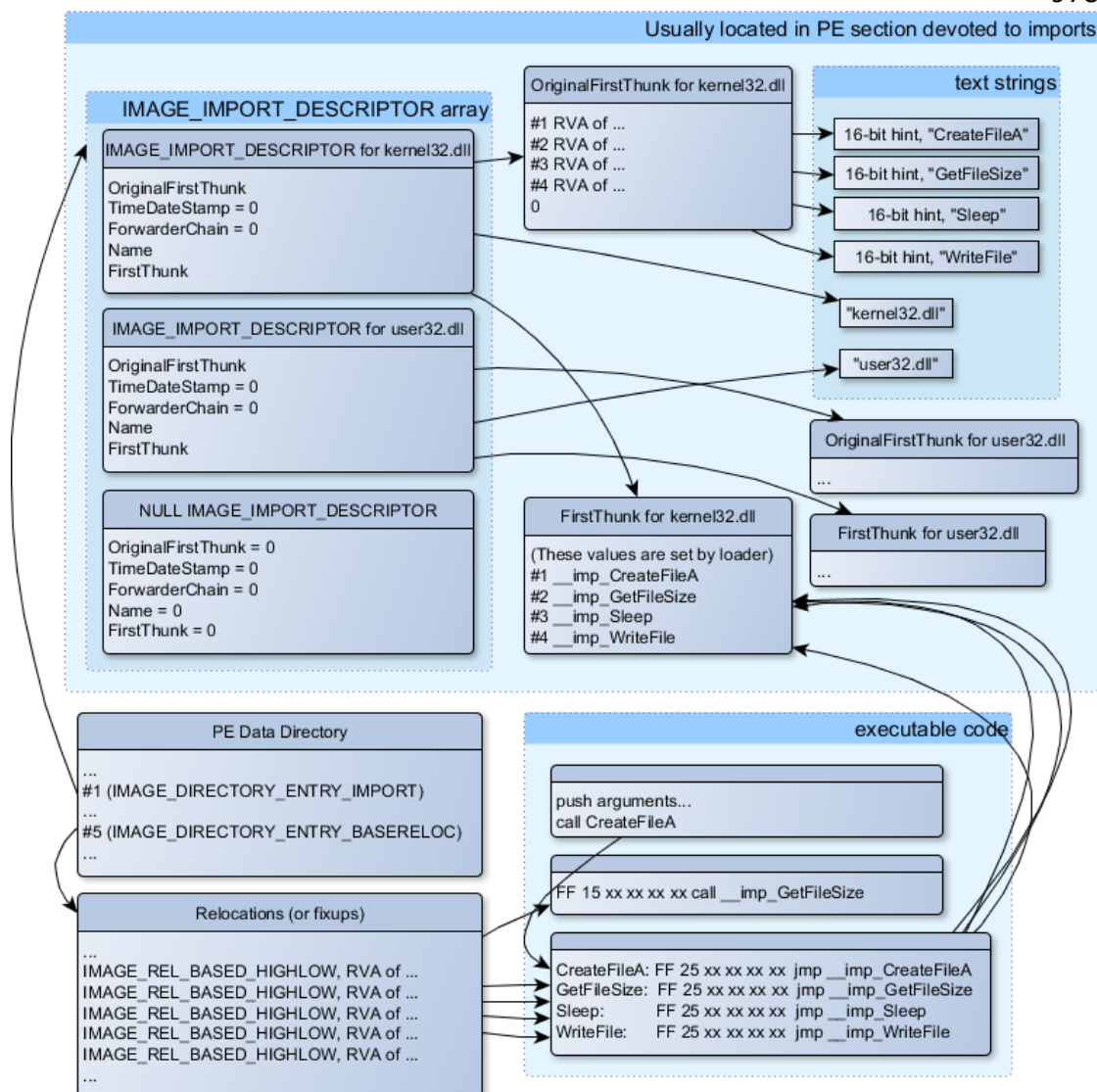


Рис. 6.1: схема, объединяющая все структуры в PE-файлы, связанные с импортами

Самая главная структура — это массив *IMAGE\_IMPORT\_DESCRIPTOR*. Каждый элемент на каждую импортируемую DLL.

У каждого элемента есть **RVA!**-адрес текстовой строки (имя DLL) (*Name*).

*OriginalFirstThunk* это **RVA!**-адрес таблицы **INT!**. Это массив **RVA!**-адресов, каждый из которых указывает на текстовую строку где записано имя функции. Каждую строку предваряет 16-битное число («hint») — «оринал» функции.

Если при загрузке удастся найти функцию по ординалу, тогда сравнение тек-

стовых строк не будет происходить. Массив оканчивается нулем.

Есть также указатель на таблицу **IAT!** с названием *FirstThunk*, это просто **RVA!**-адрес места, где загрузчик будет проставлять адреса найденных функций.

Места где загрузчик проставляет адреса, **IDA!** именуется так: `__imp_CreateFileA`, etc.

Есть по крайней мере два способа использовать адреса, проставленные загрузчиком.

- В коде будут просто инструкции вроде `call __imp_CreateFileA`, а так как, поле с адресом импортируемой функции это как бы глобальная переменная, то в таблице релоков добавляется адрес (плюс 1 или 2) в инструкции `call`, на случай если модуль будет загружен по другому базовому адресу.

Но как видно, это приводит к увеличению таблицы релоков. Ведь вызовов импортируемой функции у вас в модуле может быть очень много. К тому же, чем больше таблица релоков, тем дольше загрузка.

- На каждую импортируемую функцию выделяется только один переход на импортируемую функцию используя инструкцию `JMP` плюс релок на эту инструкцию. Такие места-«переходники» называются также «think»-ами. А все вызовы импортируемой функции это просто инструкция `CALL` на соответствующий «think». В данном случае, дополнительные релоки не нужны, потому что эти `CALL`-ы имеют относительный адрес, и корректировать их не надо.

Оба этих два метода могут комбинироваться. Надо полагать, линкер создает отдельный «think», если вызовов слишком много, но по умолчанию — не создает.

Кстати, массив адресов функций, на который указывает *FirstThunk*, не обязательно может быть в секции **IAT!**. К примеру, автор сих строк написал утилиту `PE_add_import`<sup>32</sup> для добавления импорта в уже существующий .exe-файл. Раньше, в прошлых версиях утилиты, на месте функции, вместо которой вы хотите подставить вызов в другую DLL, моя утилита вписывала такой код:

```
MOV EAX, [yourdll.dll!function]
JMP EAX
```

При этом, *FirstThunk* указывает прямо на первую инструкцию. Иными словами, загрузчик, загружая `yourdll.dll`, прописывает адрес функции *function* прямо в коде.

Надо также отметить что обычно секция кода защищена от записи, так что, моя утилита добавляет флаг `IMAGE_SCN_MEM_WRITE` для секции кода. Иначе при загрузке такой программы, она упадет с ошибкой 5 (access denied).

Может возникнуть вопрос: а что если я поставлю программу с набором DLL,

<sup>32</sup>[yurichev.com](http://yurichev.com)



которые никогда не будут меняться (в т.ч., адреса всех функций в этих DLL), может как-то можно ускорить процесс загрузки?

Да, можно прописать адреса импортируемых функций в массивы FirstThunk заранее.

Для этого в структуре *IMAGE\_IMPORT\_DESCRIPTOR* имеется поле *Timestamp*. И если там присутствует какое-то значение, то загрузчик сверяет это значение с датой-временем DLL-файла. И если они равны, то загрузчик больше ничего не делает, и загрузка может происходить быстрее.

Это называется «old-style binding»<sup>33</sup>. В Windows SDK для этого имеется утилита BIND.EXE. Для ускорения загрузки вашей программы, Matt Pietrek в Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)]<sup>34</sup>, предлагает делать binding сразу после инсталляции вашей программы на компьютере конечного пользователя.

Запаковщики/зашифровщики PE-файлов могут также сжимать/шифровать таблицу импортов. В этом случае, загрузчик Windows, конечно же, не загрузит все нужные DLL. Поэтому распаковщик/расшифровщик делает это сам, при помощи вызовов *LoadLibrary()* и *GetProcAddress()*. Вот почему в запакованных файлах эти две функции часто присутствуют в **IAT!**.

В стандартных DLL входящих в состав Windows, часто, **IAT!** находится в самом начале PE-файла.

Возможно это для оптимизации. Ведь .exe-файл при загрузке не загружается в память весь (вспомните что инсталляторы огромного размера подозрительно быстро запускаются), он «мапится» (map), и подгружается в память частями по мере обращения к этой памяти.

И возможно в Microsoft решили, что так будет быстрее.

## Ресурсы

Ресурсы в PE-файле — это набор иконок, картинок, текстовых строк, описаний диалогов. Возможно, их в свое время решили отделить от основного кода, чтобы все эти вещи были многоязычными, и было проще выбирать текст или картинку того языка, который установлен в **OS!**.

В качестве побочного эффекта, их легко редактировать и сохранять обратно в исполняемый файл, даже не обладая специальными знаниями, например, редактором ResHack (?? (стр. ??)).

## .NET

Программы на .NET компилируются не в машинный код, а в свой собственный байткод. Собственно, в .exe-файле байткод вместо обычного кода, однако, точка входа (**OEP!**) указывает на крохотный фрагмент x86-кода:

<sup>33</sup>MSDN. Существует также «new-style binding».

<sup>34</sup>Также доступно здесь: <http://msdn.microsoft.com/en-us/magazine/bb985992.aspx>

jmp	mscoree.dll!_CorExeMain
-----	-------------------------

А в mscoree.dll и находится .NET-загрузчик, который уже сам будет работать с PE-файлом.

Так было в **OS!** до Windows XP. Начиная с XP, загрузчик **OS!** уже сам определяет, что это .NET-файл и запускает его не исполняя этой инструкции JMP<sup>35</sup>.

## TLS

Эта секция содержит в себе инициализированные данные для **TLS!**(?? (стр. ??)) (если нужно). При старте нового треда, его **TLS!**-данные инициализируются данными из этой секции.

Помимо всего прочего, спецификация PE-файла предусматривает инициализацию **TLS!**-секции, т.н., TLS callbacks. Если они присутствуют, то они будут вызваны перед тем как передать управление на главную точку входа (**OEP!**). Это широко используется запаковщиками/зашифровщиками PE-файлов.

## Инструменты

- objdump (имеется в sudewin) для вывода всех структур PE-файла.
- Hiew(?? (стр. ??)) как редактор.
- pefile — Python-библиотека для работы с PE-файлами <sup>36</sup>.
- ResHack **AKA!** Resource Hacker — редактор ресурсов <sup>37</sup>.
- PE\_add\_import<sup>38</sup> — простая утилита для добавления символа/-ов в таблицу импортов PE-файла.
- PE\_patcher<sup>39</sup> — простая утилита для модификации PE-файлов.
- PE\_search\_str\_refs<sup>40</sup> — простая утилита для поиска функции в PE-файле, где используется некая текстовая строка.

## Further reading

- Daniel Pistelli — The .NET File Format <sup>41</sup>

<sup>35</sup>MSDN

<sup>36</sup><https://code.google.com/p/pefile/>

<sup>37</sup><https://code.google.com/p/pefile/>

<sup>38</sup>[http://yurichev.com/PE\\_add\\_imports.html](http://yurichev.com/PE_add_imports.html)

<sup>39</sup>yurichev.com

<sup>40</sup>yurichev.com

<sup>41</sup><http://www.codeproject.com/Articles/12585/The-.NET-File-Format>

### 6.5.3. Windows SEH

#### Забудем на время о MSVC

**SEH!** в Windows предназначен для обработки исключений, тем не менее, с Си++и **OOP!** он никак не связан. Здесь мы рассмотрим **SEH!** изолированно от Си++ и расширений MSVC.

Каждый процесс имеет цепочку **SEH!**-обработчиков, и адрес обработчика, определенного последним, записан в каждом **TIB!**. Когда происходит исключение (деление на ноль, обращение по неверному адресу в памяти, пользовательское исключение, поднятое при помощи `RaiseException()`), **OS!** находит последний обработчик в **TIB!** и вызывает его, передав ему тип исключения и всю информацию о состоянии **CPU!** в момент исключения (все значения регистров, и т. д.). Обработчик выясняет, то ли это исключение, для которого он создавался?

Если да, то он обрабатывает исключение. Если нет, то показывает **OS!** что он не может его обработать и **OS!** вызывает следующий обработчик в цепочке, и так до тех пор, пока не найдется обработчик способный обработать исключение.

В самом конце цепочки находится стандартный обработчик, показывающий всем очень известное окно, сообщающее что процесс упал, сообщает также состояние **CPU!** в момент падения и позволяет собрать и отправить информацию обработчикам в Microsoft.

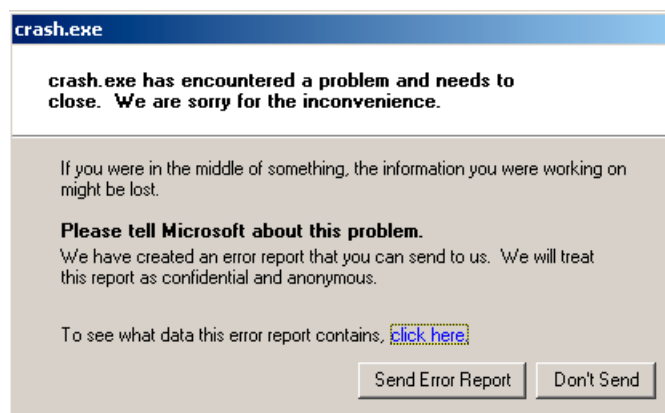


Рис. 6.2: Windows XP

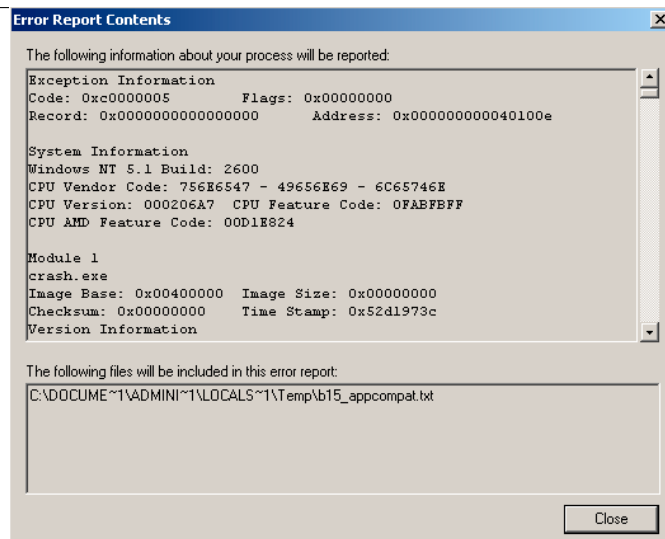


Рис. 6.3: Windows XP

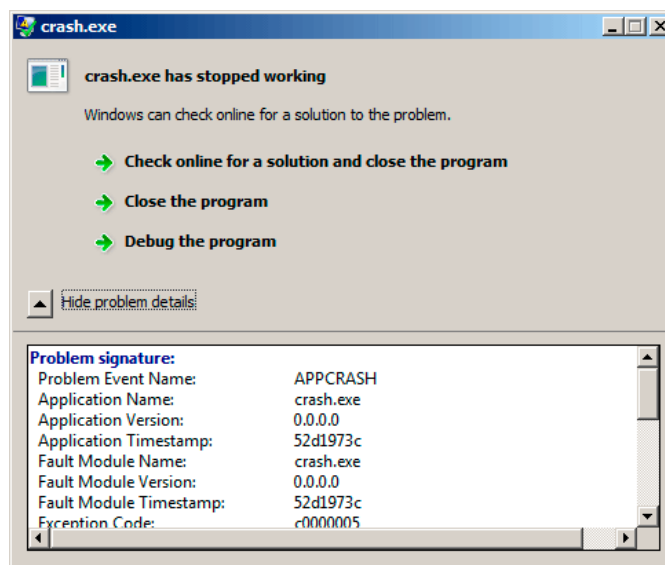


Рис. 6.4: Windows 7

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

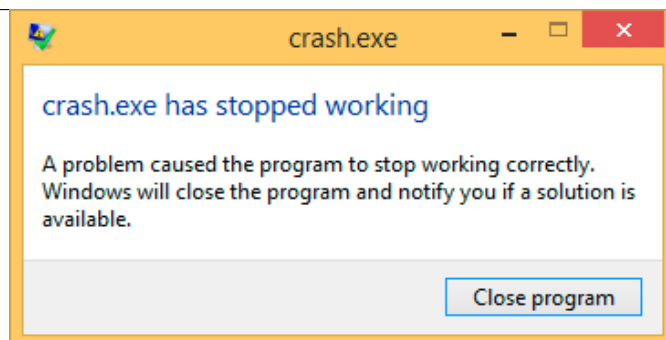


Рис. 6.5: Windows 8.1

Раньше этот обработчик назывался Dr. Watson.

Кстати, некоторые разработчики делают свой собственный обработчик, отправляющий информацию о падении программы им самим.

Он регистрируется при помощи функции `SetUnhandledExceptionFilter()` и будет вызван если **OS!** не знает, как иначе обработать исключение.

А, например, Oracle RDBMS в этом случае генерирует огромные дампы, содержащие всю возможную информацию и состоянии **CPU!** и памяти.

Попробуем написать свой примитивный обработчик исключений. Этот пример основан на примере из [Matt Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Handling*, (1997)]<sup>42</sup>. Он должен компилироваться с опцией `SAFESEH: cl seh1.cpp /link /safeseh:no`. Подробнее об опции `SAFESEH` здесь: [MSDN](#).

```
#include <windows.h>
#include <stdio.h>

DWORD new_value=1234;

EXCEPTION_DISPOSITION __cdecl except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext )
{
    unsigned i;

    printf ("%s\n", __FUNCTION__);
    printf ("ExceptionRecord->ExceptionCode=0x%p\n", ExceptionRecord->
↳ ExceptionCode);
    printf ("ExceptionRecord->ExceptionFlags=0x%p\n", ExceptionRecord->
↳ ExceptionFlags);
    printf ("ExceptionRecord->ExceptionAddress=0x%p\n", ExceptionRecord->
↳ ->ExceptionAddress);
```

<sup>42</sup>Также доступно здесь: <http://www.microsoft.com/msj/0197/Exception/Exception.aspx>

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

if (ExceptionRecord->ExceptionCode==0xE1223344)
{
    printf ("That's for us\n");
    // yes, we "handled" the exception
    return ExceptionContinueExecution;
}
else if (ExceptionRecord->ExceptionCode==EXCEPTION_ACCESS_VIOLATION
↵ )
{
    printf ("ContextRecord->Eax=0x%08X\n", ContextRecord->Eax);
    // will it be possible to 'fix' it?
    printf ("Trying to fix wrong pointer address\n");
    ContextRecord->Eax=(DWORD)&new_value;
    // yes, we "handled" the exception
    return ExceptionContinueExecution;
}
else
{
    printf ("We do not handle this\n");
    // someone else's problem
    return ExceptionContinueSearch;
};
}

int main()
{
    DWORD handler = (DWORD)except_handler; // take a pointer to our
handler

    // install exception handler
    _asm
    {
record:
        push    handler        // address of handler function
        push    FS:[0]         // address of previous handler
        mov     FS:[0],ESP     // add new EXCEPTION_REGISTRATION
    }

    RaiseException (0xE1223344, 0, 0, NULL);

    // now do something very bad
    int* ptr=NULL;
    int val=0;
    val=*ptr;
    printf ("val=%d\n", val);

    // deinstall exception handler
    _asm
    {
record
        mov     eax,[ESP]      // get pointer to previous record
        mov     FS:[0], EAX   // install previous record
        add     esp, 8         // clean our EXCEPTION_REGISTRATION
off stack
    }
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

return 0;
}

```

Сегментный регистр FS: в win32 указывает на **TIB!**. Самый первый элемент **TIB!** это указатель на последний обработчик в цепочке. Мы сохраняем его в стеке и записываем туда адрес своего обработчика. Эта структура называется `_EXCEPTION_REGISTRATION`, это простейший односвязный список, и эти элементы хранятся прямо в стеке.

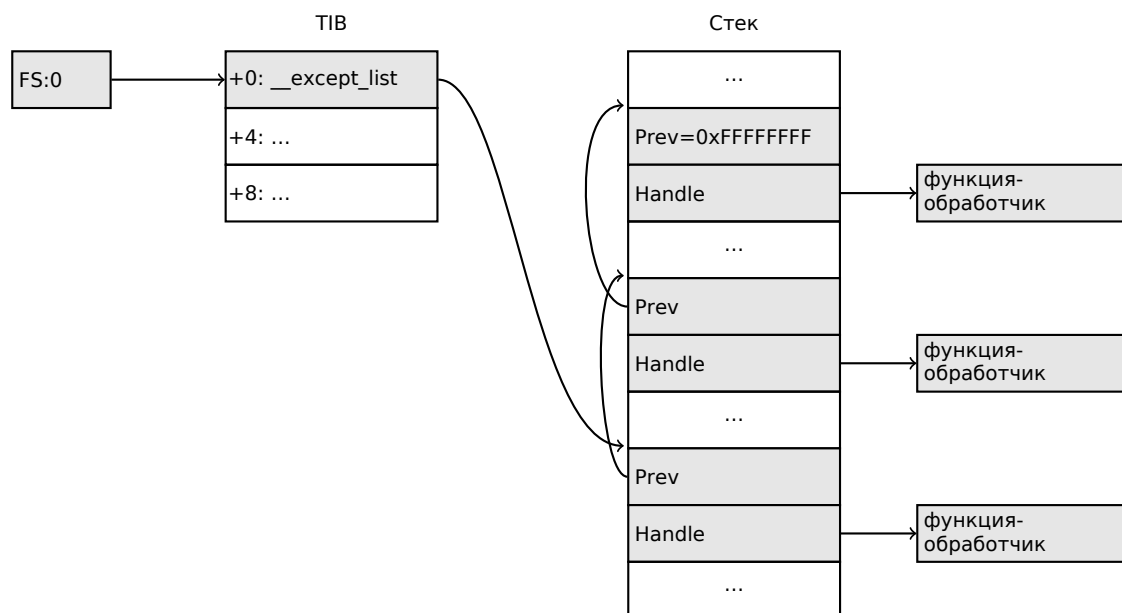
Листинг 6.22: MSVC/VC/crt/src/exsup.inc

```

_EXCEPTION_REGISTRATION struc
    prev    dd    ?
    handler dd    ?
_EXCEPTION_REGISTRATION ends

```

Так что каждое поле «handler» указывает на обработчик, а каждое поле «prev» указывает на предыдущую структуру в цепочке обработчиков. Самая последняя структура имеет `0xFFFFFFFF` (-1) в поле «prev».



После инсталляции своего обработчика, вызываем `RaiseException()`<sup>43</sup>. Это пользовательские исключения. Обработчик проверяет код.

Если код `0xE1223344`, то он возвращает `ExceptionContinueExecution`, что сигнализирует системе что обработчик скорректировал состояние CPU (обычно это регистры EIP/ESP) и что **OS!** может возобновить исполнение треда. Если вы немного измените код так что обработчик будет возвращать `ExceptionContinueSearch`, то **OS!** будет вызывать остальные обработчики в цепочке, и вряд ли найдется

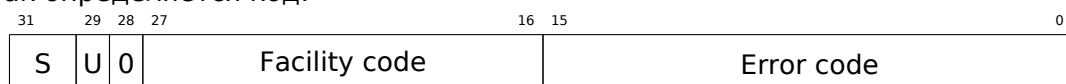
<sup>43</sup>MSDN

тот, кто обработает ваше исключение, ведь информации о нем (вернее, его коде) ни у кого нет. Вы увидите стандартное окно Windows о падении процесса.

Какова разница между системными исключениями и пользовательскими? Вот системные:

как определен в WinBase.h	как определен в ntstatus.h	как число
EXCEPTION_ACCESS_VIOLATION	STATUS_ACCESS_VIOLATION	0xC0000005
EXCEPTION_DATATYPE_MISALIGNMENT	STATUS_DATATYPE_MISALIGNMENT	0x80000002
EXCEPTION_BREAKPOINT	STATUS_BREAKPOINT	0x80000003
EXCEPTION_SINGLE_STEP	STATUS_SINGLE_STEP	0x80000004
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	STATUS_ARRAY_BOUNDS_EXCEEDED	0xC000008C
EXCEPTION_FLT_DENORMAL_OPERAND	STATUS_FLOAT_DENORMAL_OPERAND	0xC000008D
EXCEPTION_FLT_DIVIDE_BY_ZERO	STATUS_FLOAT_DIVIDE_BY_ZERO	0xC000008E
EXCEPTION_FLT_INEXACT_RESULT	STATUS_FLOAT_INEXACT_RESULT	0xC000008F
EXCEPTION_FLT_INVALID_OPERATION	STATUS_FLOAT_INVALID_OPERATION	0xC0000090
EXCEPTION_FLT_OVERFLOW	STATUS_FLOAT_OVERFLOW	0xC0000091
EXCEPTION_FLT_STACK_CHECK	STATUS_FLOAT_STACK_CHECK	0xC0000092
EXCEPTION_FLT_UNDERFLOW	STATUS_FLOAT_UNDERFLOW	0xC0000093
EXCEPTION_INT_DIVIDE_BY_ZERO	STATUS_INTEGER_DIVIDE_BY_ZERO	0xC0000094
EXCEPTION_INT_OVERFLOW	STATUS_INTEGER_OVERFLOW	0xC0000095
EXCEPTION_PRIV_INSTRUCTION	STATUS_PRIVILEGED_INSTRUCTION	0xC0000096
EXCEPTION_IN_PAGE_ERROR	STATUS_IN_PAGE_ERROR	0xC0000006
EXCEPTION_ILLEGAL_INSTRUCTION	STATUS_ILLEGAL_INSTRUCTION	0xC000001D
EXCEPTION_NONCONTINUABLE_EXCEPTION	STATUS_NONCONTINUABLE_EXCEPTION	0xC0000025
EXCEPTION_STACK_OVERFLOW	STATUS_STACK_OVERFLOW	0xC00000FD
EXCEPTION_INVALID_DISPOSITION	STATUS_INVALID_DISPOSITION	0xC0000026
EXCEPTION_GUARD_PAGE	STATUS_GUARD_PAGE_VIOLATION	0x80000001
EXCEPTION_INVALID_HANDLE	STATUS_INVALID_HANDLE	0xC0000008
EXCEPTION_POSSIBLE_DEADLOCK	STATUS_POSSIBLE_DEADLOCK	0xC0000194
CONTROL_C_EXIT	STATUS_CONTROL_C_EXIT	0xC000013A

Так определяется код:



S это код статуса: 11 — ошибка; 10 — предупреждение; 01 — информация; 00 — успех. U — является ли этот код пользовательским, а не системным.

Вот почему мы выбрали 0xE1223344 — E<sub>16</sub> (1110<sub>2</sub>) 0xE (1110<sub>b</sub>) означает, что это 1) пользовательское исключение; 2) ошибка. Хотя, если быть честным, этот пример нормально работает и без этих старших бит.

Далее мы пытаемся прочитать значение из памяти по адресу 0. Конечно, в win32 по этому адресу обычно ничего нет, и сработает исключение. Однако, первый обработчик, который будет заниматься этим делом — ваш, и он узнает об этом первым, проверяя код на соответствие с константной EXCEPTION\_ACCESS\_VIOLATION.

А если заглянуть в то что получилось на ассемблере, то можно увидеть, что код читающий из памяти по адресу 0, выглядит так:

#### Листинг 6.23: MSVC 2010

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

...
xor    eax, eax
mov    eax, DWORD PTR [eax] ; exception will occur here
push  eax
push  OFFSET msg
call  _printf
add   esp, 8
...

```

Возможно ли «на лету» исправить ошибку и предложить программе исполняться далее? Да, наш обработчик может изменить значение в EAX и предложить **OS!** исполнить эту же инструкцию еще раз. Что мы и делаем. `printf()` напечатает 1234, потому что после работы нашего обработчика, EAX будет не 0, а будет содержать адрес глобальной переменной `new_value`. Программа будет исполняться далее.

Собственно, вот что происходит: срабатывает защита менеджера памяти в **CPU!**, он останавливает работу треда, отыскивает в ядре Windows обработчик исключений, тот, в свою очередь, начинает вызывать обработчики из цепочки **SEH!**, по одному.

Мы компилируем это всё в MSVC 2010, но конечно же, нет никакой гарантии что для указателя будет использован именно регистр EAX.

Этот трюк с подменой адреса эффектно выглядит, и мы рассматриваем его здесь для наглядной иллюстрации работы **SEH!**.

Тем не менее, трудно припомнить, применяется ли где-то подобное на практике для исправления ошибок «на лету».

Почему SEH-записи хранятся именно в стеке а не в каком-то другом месте? Возможно, потому что **OS!** не нужно заботиться об освобождении этой информации, эти записи просто пропадают как ненужные когда функция заканчивает работу.

Это чем-то похоже на `alloca(): (?? (стр. ??))`.

### Теперь вспомним MSVC

Должно быть, программистам Microsoft были нужны исключения в Си, но не в Си++(для использования в ядре Windows NT, которое написано на Си), так что они добавили нестандартное расширение Си в MSVC <sup>44</sup>. Оно не связано с исключениями в Си++.

```

__try
{
    ...
}
__except(filter code)
{
    handler code
}

```

<sup>44</sup>MSDN

Блок «finally» может присутствовать вместо код обработчика:

```

__try
{
    ...
}
__finally
{
    ...
}

```

Код-фильтр — это выражение, отвечающее на вопрос, соответствует ли код этого обработчика к поднятому исключению. Если ваш код слишком большой и не помещается в одно выражение, отдельная функция-фильтр может быть определена.

Таких конструкций много в ядре Windows. Вот несколько примеров оттуда (**WRK!**):

Листинг 6.24: WRK-v1.2/base/ntos/ob/obwait.c

```

try {
    KeReleaseMutant( (PKMUTANT)SignalObject,
                    MUTANT_INCREMENT,
                    FALSE,
                    TRUE );
} except((GetExceptionCode () == STATUS_ABANDONED ||
        GetExceptionCode () == STATUS_MUTANT_NOT_OWNED)?
        EXCEPTION_EXECUTE_HANDLER :
        EXCEPTION_CONTINUE_SEARCH) {
    Status = GetExceptionCode();
    goto WaitExit;
}

```

Листинг 6.25: WRK-v1.2/base/ntos/cache/cachesub.c

```

try {
    RtlCopyBytes( (PVOID)((PCHAR)CacheBuffer + PageOffset),
                UserBuffer,
                MorePages ?
                (PAGE_SIZE - PageOffset) :
                (ReceivedLength - PageOffset) );
} except( CcCopyReadExceptionFilter( GetExceptionInformation(),
                                    &Status ) ) {

```

Вот пример кода-фильтра:

Листинг 6.26: WRK-v1.2/base/ntos/cache/copysup.c

```
LONG
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

CcCopyReadExceptionFilter(
    IN PEXCEPTION_POINTERS ExceptionPointer,
    IN PNTSTATUS ExceptionCode
)

/**+
Routine Description:

    This routine serves as an exception filter and has the special job of
    extracting the "real" I/O error when Mm raises STATUS_IN_PAGE_ERROR
    beneath us.

Arguments:

    ExceptionPointer - A pointer to the exception record that contains
        the real Io Status.

    ExceptionCode - A pointer to an NTSTATUS that is to receive the real
        status.

Return Value:

    EXCEPTION_EXECUTE_HANDLER

--*/
{
    *ExceptionCode = ExceptionPointer->ExceptionRecord->ExceptionCode;

    if ( (*ExceptionCode == STATUS_IN_PAGE_ERROR) &&
        (ExceptionPointer->ExceptionRecord->NumberParameters >= 3) ) {

        *ExceptionCode = (NTSTATUS) ExceptionPointer->ExceptionRecord->
        ↵ ExceptionInformation[2];
    }

    ASSERT( !NT_SUCCESS(*ExceptionCode) );

    return EXCEPTION_EXECUTE_HANDLER;
}

```

Внутри, SEH это расширение исключений поддерживаемых OS.

Но функция обработчик теперь или `_except_handler3` (для SEH3) или `_except_handler4` (для SEH4). Код обработчика от MSVC, расположен в его библиотеках, или же в `msvc*.dll`. Очень важно понимать, что SEH это специфичное для MSVC. Другие win32-компиляторы могут предлагать что-то совершенно другое.

### SEH3

SEH3 имеет `_except_handler3` как функцию-обработчик, и расширяет структу-

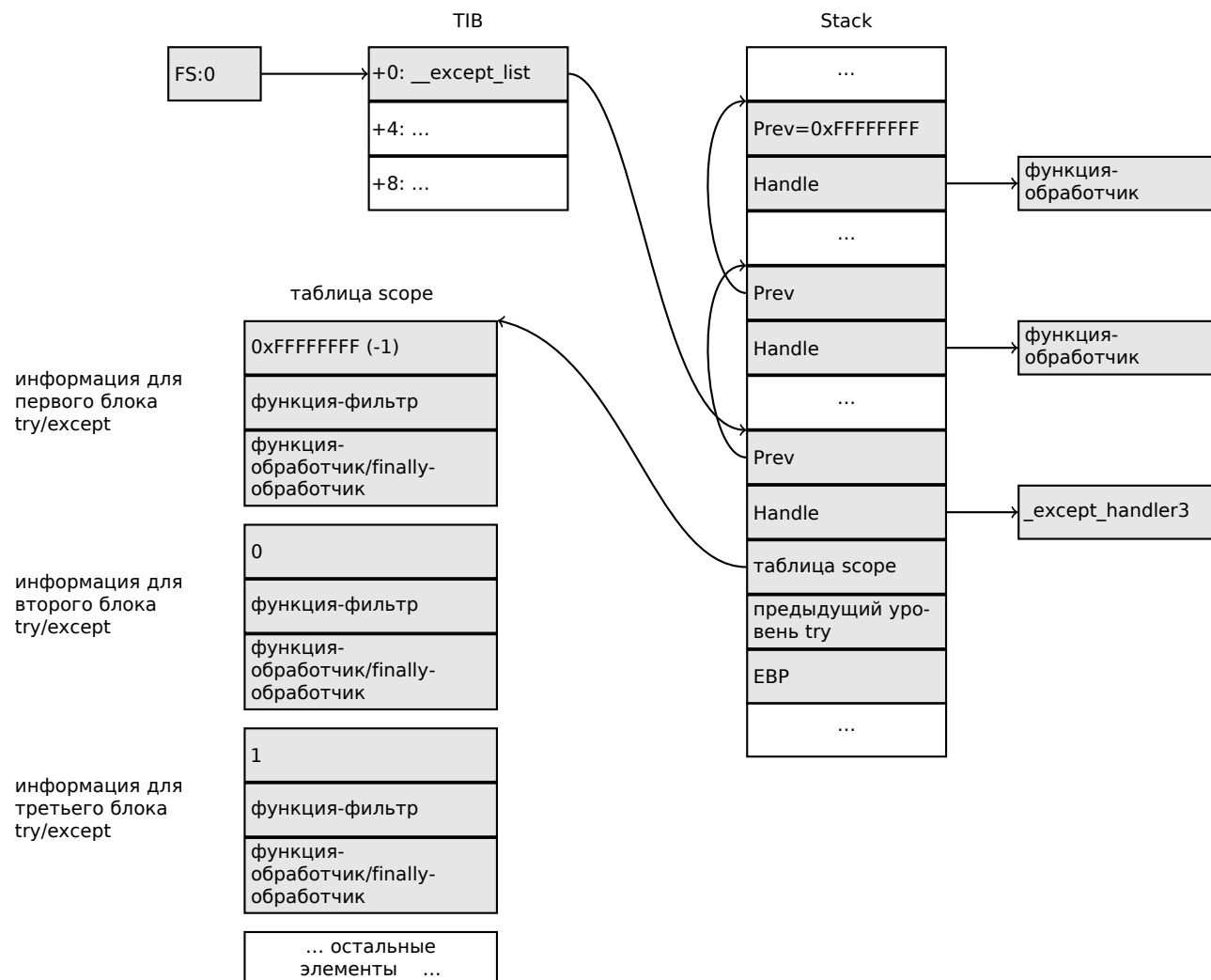
---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

ру

`_EXCEPTION_REGISTRATION` добавляя указатель на *scope table* и переменную *previous try level*. SEH4 расширяет *scope table* добавляя еще 4 значения связанных с защитой от переполнения буфера.

*scope table* это таблица, состоящая из указателей на код фильтра и обработчика, для каждого уровня вложенности *try/except*.



И снова, очень важно понимать, что OS заботится только о полях *prev/handle*, и больше ничего. Это работа функции `_except_handler3` читать другие поля, читать *scope table* и решать, какой обработчик исполнять и когда.

Исходный код функции `_except_handler3` закрыт. Хотя, Sanos OS, имеющая слой совместимости с win32, имеет некоторые функции написанные заново,

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

которые в каком-то смысле эквивалентны тем что в Windows<sup>45</sup>. Другие попытки реализации имеются в Wine<sup>46</sup> и ReactOS<sup>47</sup>.

Если указатель *filter* ноль, *handler* указывает на код *finally*.

Во время исполнения, значение *previous try level* в стеке меняется, чтобы функция `_except_handler3` знала о текущем уровне вложенности, чтобы знать, какой элемент таблицы *scope table* использовать.

### СЕНЗ: пример с одним блоком try/except

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int main()
{
    int* p = NULL;
    __try
    {
        printf("hello #1!\n");
        *p = 13;    // causes an access violation exception;
        printf("hello #2!\n");
    }
    __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
             EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
        printf("access violation, can't recover\n");
    }
}
```

Листинг 6.27: MSVC 2003

```
$SG74605 DB    'hello #1!', 0aH, 00H
$SG74606 DB    'hello #2!', 0aH, 00H
$SG74608 DB    'access violation, can't recover', 0aH, 00H
_DATA    ENDS

; scope table:
CONST    SEGMENT
$T74622  DD    0ffffffffH    ; previous try level
         DD    FLAT:$L74617 ; filter
         DD    FLAT:$L74618 ; handler

CONST    ENDS
_TEXT    SEGMENT
$T74621 = -32 ; size = 4
_p$ = -28   ; size = 4
```

<sup>45</sup><https://code.google.com/p/sanos/source/browse/src/win32/msvcrt/except.c>

<sup>46</sup>GitHub

<sup>47</sup>[http://doxygen.reactos.org/d4/df2/lib\\_2sdk\\_2crt\\_2except\\_2except\\_8c\\_source.html](http://doxygen.reactos.org/d4/df2/lib_2sdk_2crt_2except_2except_8c_source.html)

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

__$SEHRec$ = -24 ; size = 24
_main PROC NEAR
    push    ebp
    mov     ebp, esp
    push    -1 ; previous try level
    push    OFFSET FLAT:$T74622 ; scope table
    push    OFFSET FLAT:__except_handler3 ; handler
    mov     eax, DWORD PTR fs:__except_list
    push    eax ; prev
    mov     DWORD PTR fs:__except_list, esp
    add     esp, -16
; 3 registers to be saved:
    push    ebx
    push    esi
    push    edi
    mov     DWORD PTR __SEHRec$[ebp], esp
    mov     DWORD PTR _p$[ebp], 0
    mov     DWORD PTR __SEHRec$[ebp+20], 0 ; previous try level
    push    OFFSET FLAT:$SG74605 ; 'hello #1!'
    call   _printf
    add     esp, 4
    mov     eax, DWORD PTR _p$[ebp]
    mov     DWORD PTR [eax], 13
    push    OFFSET FLAT:$SG74606 ; 'hello #2!'
    call   _printf
    add     esp, 4
    mov     DWORD PTR __SEHRec$[ebp+20], -1 ; previous try level
    jmp    SHORT $L74616

; filter code:
$L74617:
$L74627:
    mov     ecx, DWORD PTR __SEHRec$[ebp+4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR [edx]
    mov     DWORD PTR $T74621[ebp], eax
    mov     eax, DWORD PTR $T74621[ebp]
    sub     eax, -1073741819; c0000005H
    neg     eax
    sbb     eax, eax
    inc     eax
$L74619:
$L74626:
    ret     0

; handler code:
$L74618:
    mov     esp, DWORD PTR __SEHRec$[ebp]
    push    OFFSET FLAT:$SG74608 ; 'access violation, can't recover'
    call   _printf
    add     esp, 4
    mov     DWORD PTR __SEHRec$[ebp+20], -1 ; setting previous try level back
to -1
$L74616:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

xor    eax, eax
mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
mov    DWORD PTR fs:__except_list, ecx
pop    edi
pop    esi
pop    ebx
mov    esp, ebp
pop    ebp
ret    0
_main  ENDP
_TEXT  ENDS
END

```

Здесь мы видим, как структура SEH конструируется в стеке. *Scope table* расположена в сегменте CONST — действительно, эти поля не будут меняться. Интересно, как меняется переменная *previous try level*. Исходное значение 0xFFFFFFFF (-1). Момент, когда тело try открывается, обозначен инструкцией, записывающей 0 в эту переменную. В момент, когда тело try закрывается, -1 возвращается в нее назад. Мы также видим адреса кода фильтра и обработчика. Так мы можем легко увидеть структуру конструкций *try/except* в функции.

Так как код инициализации SEH-структур в прологе функций может быть общим для нескольких функций, иногда компилятор вставляет в прологе вызов функции `SEH_prolog()`, которая всё это делает. А код для деинициализации SEH в функции `SEH_epilog()`.

Запустим этот пример в [tracer](#):

```
tracer.exe -l:2.exe --dump-seh
```

Листинг 6.28: tracer.exe output

```

EXCEPTION_ACCESS_VIOLATION at 2.exe!main+0x44 (0x401054) ↵
↳ ExceptionInformation[0]=1
EAX=0x00000000 EBX=0x7efde000 ECX=0x0040cbc8 EDX=0x00008e3c8
ESI=0x00001db1 EDI=0x00000000 EBP=0x0018feac ESP=0x0018fe80
EIP=0x00401054
FLAGS=AF IF RF
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x401204 (2.exe!↵
↳ __except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401070 (2.exe!main+0↵
↳ x60) handler=0x401088 (2.exe!main+0x78)
* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x401204 (2.exe!↵
↳ __except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401531 (2.exe!↵
↳ mainCRTStartup+0x18d) handler=0x401545 (2.exe!mainCRTStartup+0x1a1)
* SEH frame at 0x18ffc4 prev=0x18ffe4 handler=0x771f71f5 (ntdll.dll!↵
↳ __except_handler4)
SEH4 frame. previous trylevel=0
SEH4 header:    GSCookieOffset=0xffffffff GSCookieXOROffset=0x0

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

EHCookieOffset=0xffffffff EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (ntdll.dll!↵
↳ __safe_se_handler_table+0x20) handler=0x771f90eb (ntdll.dll!↵
↳ _TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdll.dll!↵
↳ _FinalExceptionHandler@16)

```

Мы видим, что цепочка SEH состоит из 4-х обработчиков.

Первые два расположены в нашем примере. Два? Но ведь мы же сделали только один? Да, второй был установлен в **CRT!**-функции `_mainCRTStartup()`, и судя по всему, он обрабатывает как минимум исключения связанные с **FPU!**. Его код можно посмотреть в инсталляции MSVC: `crt/src/winxfldr.c`.

Третий это SEH4 в `ntdll.dll`, и четвертый это обработчик, не имеющий отношения к MSVC, расположенный в `ntdll.dll`, имеющий «говорящее» название функции.

Как видно, в цепочке присутствуют обработчики трех типов: один не связан с MSVC вообще (последний) и два связанных с MSVC: SEH3 и SEH4.

### SEH3: пример с двумя блоками try/except

```

#include <stdio.h>
#include <windows.h>
#include <except.h>

int filter_user_exceptions (unsigned int code, struct _EXCEPTION_POINTERS *↵
↳ ep)
{
    printf("in filter. code=0x%08X\n", code);
    if (code == 0x112233)
    {
        printf("yes, that is our exception\n");
        return EXCEPTION_EXECUTE_HANDLER;
    }
    else
    {
        printf("not our exception\n");
        return EXCEPTION_CONTINUE_SEARCH;
    }
};

int main()
{
    int* p = NULL;
    _try
    {
        _try
        {
            printf ("hello!\n");
            RaiseException (0x112233, 0, 0, NULL);
        }
    }
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

        printf ("0x112233 raised. now let's crash\n");
        *p = 13;    // causes an access violation exception;
    }
    __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
             EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
        printf("access violation, can't recover\n");
    }
}
__except(filter_user_exceptions(GetExceptionCode(),
↵ GetExceptionInformation()))
{
    // the filter_user_exceptions() function answering to the question
    // "is this exception belongs to this block?"
    // if yes, do the follow:
    printf("user exception caught\n");
}
}

```

Теперь здесь два блока `try`. Так что *scope table* теперь содержит два элемента, один элемент на каждый блок. *Previous try level* меняется вместе с тем, как исполнение доходит до очередного `try`-блока, либо выходит из него.

#### Листинг 6.29: MSVC 2003

```

$SG74606 DB    'in filter. code=0x%08X', 0aH, 00H
$SG74608 DB    'yes, that is our exception', 0aH, 00H
$SG74610 DB    'not our exception', 0aH, 00H
$SG74617 DB    'hello!', 0aH, 00H
$SG74619 DB    '0x112233 raised. now let's crash', 0aH, 00H
$SG74621 DB    'access violation, can't recover', 0aH, 00H
$SG74623 DB    'user exception caught', 0aH, 00H

_code$ = 8    ; size = 4
_ep$ = 12    ; size = 4
_filter_user_exceptions PROC NEAR
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _code$[ebp]
    push    eax
    push    OFFSET FLAT:$SG74606 ; 'in filter. code=0x%08X'
    call    _printf
    add     esp, 8
    cmp     DWORD PTR _code$[ebp], 1122867; 00112233H
    jne     SHORT $L74607
    push    OFFSET FLAT:$SG74608 ; 'yes, that is our exception'
    call    _printf
    add     esp, 4
    mov     eax, 1
    jmp     SHORT $L74605
$L74607:
    push    OFFSET FLAT:$SG74610 ; 'not our exception'
    call    _printf
    add     esp, 4

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    xor    eax, eax
$L74605:
    pop    ebp
    ret    0
_filter_user_exceptions ENDP

; scope table:
CONST    SEGMENT
$T74644  DD    0fffffffH    ; previous try level for outer block
         DD    FLAT:$L74634 ; outer block filter
         DD    FLAT:$L74635 ; outer block handler
         DD    00H        ; previous try level for inner block
         DD    FLAT:$L74638 ; inner block filter
         DD    FLAT:$L74639 ; inner block handler
CONST    ENDS

$T74643 = -36    ; size = 4
$T74642 = -32    ; size = 4
_p$ = -28       ; size = 4
__$SEHRec$ = -24 ; size = 24
_main    PROC NEAR
    push  ebp
    mov   ebp, esp
    push -1 ; previous try level
    push OFFSET FLAT:$T74644
    push OFFSET FLAT:__except_handler3
    mov   eax, DWORD PTR fs:__except_list
    push  eax
    mov   DWORD PTR fs:__except_list, esp
    add  esp, -20
    push  ebx
    push  esi
    push  edi
    mov   DWORD PTR __SEHRec$[ebp], esp
    mov   DWORD PTR _p$[ebp], 0
    mov   DWORD PTR __SEHRec$[ebp+20], 0 ; outer try block entered. set
previous try level to 0
    mov   DWORD PTR __SEHRec$[ebp+20], 1 ; inner try block entered. set
previous try level to 1
    push  OFFSET FLAT:$SG74617 ; 'hello!'
    call  _printf
    add  esp, 4
    push  0
    push  0
    push  0
    push  1122867    ; 00112233H
    call  DWORD PTR __imp__RaiseException@16
    push  OFFSET FLAT:$SG74619 ; '0x112233 raised. now let's crash'
    call  _printf
    add  esp, 4
    mov   eax, DWORD PTR _p$[ebp]
    mov   DWORD PTR [eax], 13
    mov   DWORD PTR __SEHRec$[ebp+20], 0 ; inner try block exited. set
previous try level back to 0

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    jmp     SHORT $L74615

; inner block filter:
$L74638:
$L74650:
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR [edx]
    mov     DWORD PTR $T74643[ebp], eax
    mov     eax, DWORD PTR $T74643[ebp]
    sub     eax, -1073741819; c0000005H
    neg     eax
    sbb    eax, eax
    inc     eax
$L74640:
$L74648:
    ret     0

; inner block handler:
$L74639:
    mov     esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET FLAT:$SG74621 ; 'access violation, can't recover'
    call   _printf
    add    esp, 4
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; inner try block exited. set
previous try level back to 0

$L74615:
    mov     DWORD PTR __$SEHRec$[ebp+20], -1 ; outer try block exited, set
previous try level back to -1
    jmp     SHORT $L74633

; outer block filter:
$L74634:
$L74651:
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR [edx]
    mov     DWORD PTR $T74642[ebp], eax
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    push   ecx
    mov     edx, DWORD PTR $T74642[ebp]
    push   edx
    call   _filter_user_exceptions
    add    esp, 8
$L74636:
$L74649:
    ret     0

; outer block handler:
$L74635:
    mov     esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET FLAT:$SG74623 ; 'user exception caught'
    call   _printf

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    add     esp, 4
    mov     DWORD PTR __$SEHRec$[ebp+20], -1 ; both try blocks exited. set
previous try level back to -1
$L74633:
    xor     eax, eax
    mov     ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov     DWORD PTR fs:__except_list, ecx
    pop     edi
    pop     esi
    pop     ebx
    mov     esp, ebp
    pop     ebp
    ret     0
_main     ENDP

```

Если установить точку останова на функцию `printf()` вызываемую из обработчика, мы можем увидеть, что добавился еще один SEH-обработчик. Наверное, это еще какая-то дополнительная механика, скрытая внутри процесса обработки исключений. Тут мы также видим *scope table* состоящую из двух элементов.

```
tracer.exe -l:3.exe bpx=3.exe!printf --dump-seh
```

#### Листинг 6.30: tracer.exe output

```

(0) 3.exe!printf
EAX=0x0000001b EBX=0x00000000 ECX=0x0040cc58 EDX=0x00008e3c8
ESI=0x00000000 EDI=0x00000000 EBP=0x0018f840 ESP=0x0018f838
EIP=0x004011b6
FLAGS=PF ZF IF
* SEH frame at 0x18f88c prev=0x18fe9c handler=0x771db4ad (ntdll.dll!
↳ ExecuteHandler2@20+0x3a)
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x4012e0 (3.exe!
↳ __except_handler3)
SEH3 frame. previous trylevel=1
scopetable entry[0]. previous try level=-1, filter=0x401120 (3.exe!main+0
↳ xb0) handler=0x40113b (3.exe!main+0xcb)
scopetable entry[1]. previous try level=0, filter=0x4010e8 (3.exe!main+0x78
↳ ) handler=0x401100 (3.exe!main+0x90)
* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x4012e0 (3.exe!
↳ __except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x40160d (3.exe!
↳ mainCRTStartup+0x18d) handler=0x401621 (3.exe!mainCRTStartup+0x1a1)
* SEH frame at 0x18ffc4 prev=0x18ffe4 handler=0x771f71f5 (ntdll.dll!
↳ __except_handler4)
SEH4 frame. previous trylevel=0
SEH4 header:      GSCookieOffset=0xffffffff GSCookieXOR0ffset=0x0
                  EHCookieOffset=0xffffffff EHCookieXOR0ffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (ntdll.dll!
↳ __safe_se_handler_table+0x20) handler=0x771f90eb (ntdll.dll!
↳ _TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdll.dll!
↳ _FinalExceptionHandler@16)

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

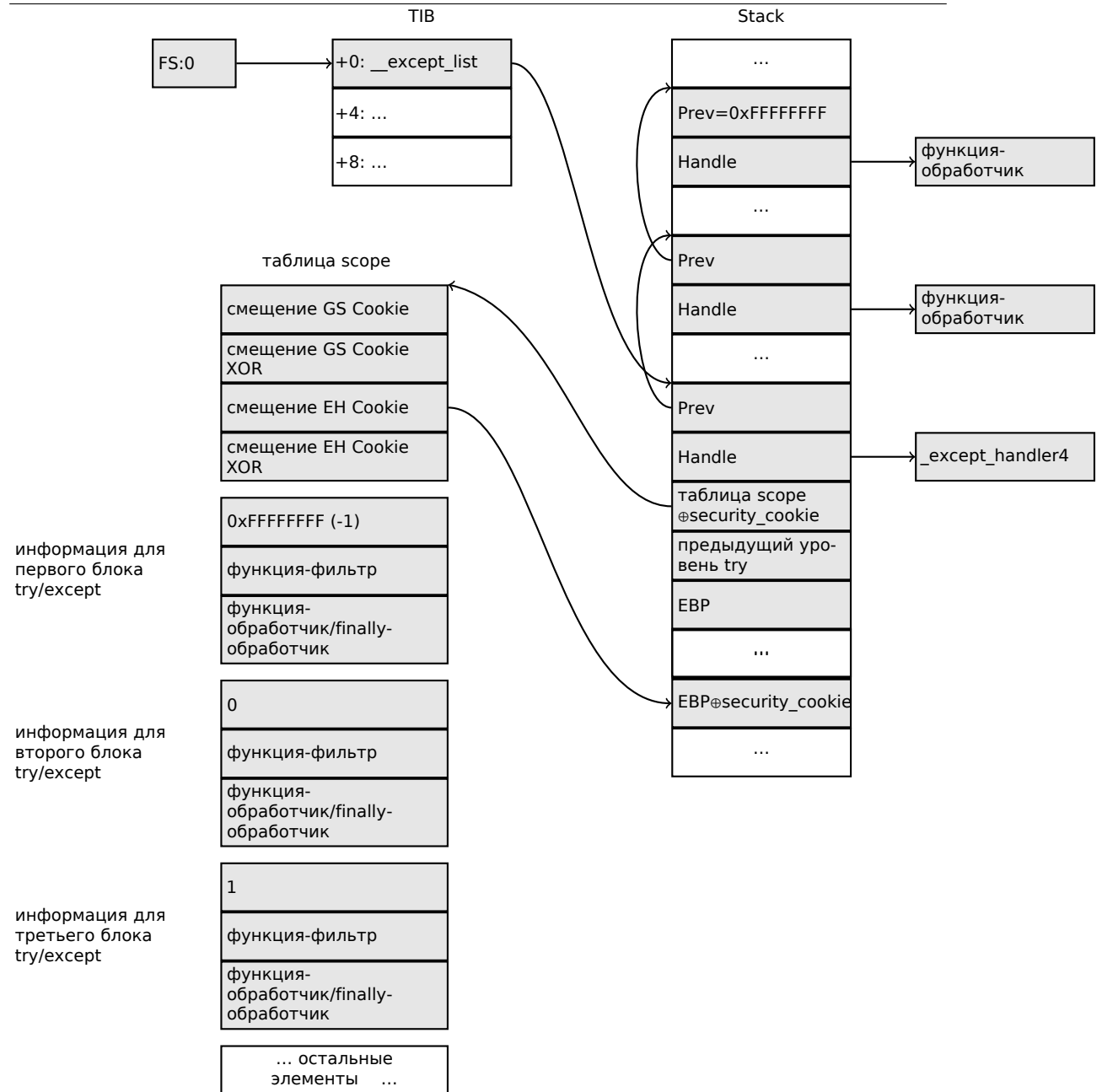
## SEH4

Во время атаки переполнения буфера (?? (стр. ??)) адрес *scope table* может быть перезаписан, так что начиная с MSVC 2005, SEH3 был дополнен защитой от переполнения буфера, до SEH4. Указатель на *scope table* теперь [про-XOR-ен](#) с [security cookie](#).

*Scope table* расширена, теперь имеет заголовок, содержащий 2 указателя на *security cookies*. Каждый элемент имеет смещение внутри стека на другое значение: это адрес [фрейма](#) (EBP) также [про-XOR-енный](#) с *security\_cookie* расположенный в стеке. Это значение будет прочитано во время обработки исключения и проверено на правильность.

*Security cookie* в стеке случайное каждый раз, так что атакующий, как мы надеемся, не может предсказать его.

Изначальное значение *previous try level* это -2 в SEH4 вместо -1.



Оба примера скомпилированы в MSVC 2012 с SEH4:

Листинг 6.31: MSVC 2012: one try block example

```
$SG85485 DB 'hello #1!', 0aH, 00H
$SG85486 DB 'hello #2!', 0aH, 00H
$SG85488 DB 'access violation, can't recover', 0aH, 00H

; scope table:
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

xdata$x          SEGMENT
__sehtable$_main DD 0fffffffEH ; GS Cookie Offset
                DD 00H          ; GS Cookie XOR Offset
                DD 0fffffffccH ; EH Cookie Offset
                DD 00H          ; EH Cookie XOR Offset
                DD 0fffffffEH ; previous try level
                DD FLAT:$LN12@main ; filter
                DD FLAT:$LN8@main ; handler
xdata$x          ENDS

$T2 = -36        ; size = 4
_p$ = -32        ; size = 4
tv68 = -28       ; size = 4
__$SEHRec$ = -24 ; size = 24
_main PROC
    push    ebp
    mov     ebp, esp
    push    -2
    push    OFFSET __sehtable$_main
    push    OFFSET __except_handler4
    mov     eax, DWORD PTR fs:0
    push    eax
    add     esp, -20
    push    ebx
    push    esi
    push    edi
    mov     eax, DWORD PTR __security_cookie
    xor     DWORD PTR __SEHRec$[ebp+16], eax ; xored pointer to scope table
    xor     eax, ebp
    push    eax ; ebp ^ security_cookie
    lea    eax, DWORD PTR __SEHRec$[ebp+8] ;
    pointer to VC_EXCEPTION_REGISTRATION_RECORD
    mov     DWORD PTR fs:0, eax
    mov     DWORD PTR __SEHRec$[ebp], esp
    mov     DWORD PTR _p$[ebp], 0
    mov     DWORD PTR __SEHRec$[ebp+20], 0 ; previous try level
    push    OFFSET $SG85485 ; 'hello #1!'
    call   _printf
    add     esp, 4
    mov     eax, DWORD PTR _p$[ebp]
    mov     DWORD PTR [eax], 13
    push    OFFSET $SG85486 ; 'hello #2!'
    call   _printf
    add     esp, 4
    mov     DWORD PTR __SEHRec$[ebp+20], -2 ; previous try level
    jmp     SHORT $LN6@main

; filter:
$LN7@main:
$LN12@main:
    mov     ecx, DWORD PTR __SEHRec$[ebp+4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR [edx]
    mov     DWORD PTR $T2[ebp], eax

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    cmp     DWORD PTR $T2[ebp], -1073741819 ; c0000005H
    jne     SHORT $LN4@main
    mov     DWORD PTR tv68[ebp], 1
    jmp     SHORT $LN5@main
$LN4@main:
    mov     DWORD PTR tv68[ebp], 0
$LN5@main:
    mov     eax, DWORD PTR tv68[ebp]
$LN9@main:
$LN11@main:
    ret     0

; handler:
$LN8@main:
    mov     esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET $SG85488 ; 'access violation, can't recover'
    call   _printf
    add     esp, 4
    mov     DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
$LN6@main:
    xor     eax, eax
    mov     ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov     DWORD PTR fs:0, ecx
    pop     ecx
    pop     edi
    pop     esi
    pop     ebx
    mov     esp, ebp
    pop     ebp
    ret     0
_main     ENDP

```

Листинг 6.32: MSVC 2012: two try blocks example

```

$SG85486 DB 'in filter. code=0x%08X', 0aH, 00H
$SG85488 DB 'yes, that is our exception', 0aH, 00H
$SG85490 DB 'not our exception', 0aH, 00H
$SG85497 DB 'hello!', 0aH, 00H
$SG85499 DB '0x112233 raised. now let's crash', 0aH, 00H
$SG85501 DB 'access violation, can't recover', 0aH, 00H
$SG85503 DB 'user exception caught', 0aH, 00H

xdata$x     SEGMENT
__sehables_main DD 0fffffffEH ; GS Cookie Offset
               DD 00H ; GS Cookie XOR Offset
               DD 0fffffffC8H ; EH Cookie Offset
               DD 00H ; EH Cookie Offset
               DD 0fffffffEH ; previous try level for outer block
               DD FLAT:$LN19@main ; outer block filter
               DD FLAT:$LN9@main ; outer block handler
               DD 00H ; previous try level for inner block
               DD FLAT:$LN18@main ; inner block filter
               DD FLAT:$LN13@main ; inner block handler
xdata$x     ENDS

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

$T2 = -40          ; size = 4
$T3 = -36          ; size = 4
_p$ = -32          ; size = 4
tv72 = -28         ; size = 4
__$SEHRec$ = -24  ; size = 24
_main PROC
    push    ebp
    mov     ebp, esp
    push    -2 ; initial previous try level
    push    OFFSET __sehtable$_main
    push    OFFSET __except_handler4
    mov     eax, DWORD PTR fs:0
    push    eax ; prev
    add     esp, -24
    push    ebx
    push    esi
    push    edi
    mov     eax, DWORD PTR __security_cookie
    xor     DWORD PTR __SEHRec$[ebp+16], eax ; xored pointer to scope
table
    xor     eax, ebp ; ebp ^ security_cookie
    push    eax
    lea    eax, DWORD PTR __SEHRec$[ebp+8] ;
pointer to VC_EXCEPTION_REGISTRATION_RECORD
    mov     DWORD PTR fs:0, eax
    mov     DWORD PTR __SEHRec$[ebp], esp
    mov     DWORD PTR _p$[ebp], 0
    mov     DWORD PTR __SEHRec$[ebp+20], 0 ; entering outer try block,
setting previous try level=0
    mov     DWORD PTR __SEHRec$[ebp+20], 1 ; entering inner try block,
setting previous try level=1
    push    OFFSET $SG85497 ; 'hello!'
    call   _printf
    add     esp, 4
    push    0
    push    0
    push    0
    push    1122867 ; 00112233H
    call   DWORD PTR __imp__RaiseException@16
    push    OFFSET $SG85499 ; '0x112233 raised. now let''s crash'
    call   _printf
    add     esp, 4
    mov     eax, DWORD PTR _p$[ebp]
    mov     DWORD PTR [eax], 13
    mov     DWORD PTR __SEHRec$[ebp+20], 0 ; exiting inner try block, set
previous try level back to 0
    jmp     SHORT $LN2@main

; inner block filter:
$LN12@main:
$LN18@main:
    mov     ecx, DWORD PTR __SEHRec$[ebp+4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR [edx]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov     DWORD PTR $T3[ebp], eax
cmp     DWORD PTR $T3[ebp], -1073741819 ; c0000005H
jne     SHORT $LN5@main
mov     DWORD PTR tv72[ebp], 1
jmp     SHORT $LN6@main
$LN5@main:
mov     DWORD PTR tv72[ebp], 0
$LN6@main:
mov     eax, DWORD PTR tv72[ebp]
$LN14@main:
$LN16@main:
ret     0

; inner block handler:
$LN13@main:
mov     esp, DWORD PTR __$SEHRec$[ebp]
push   OFFSET $SG85501 ; 'access violation, can't recover'
call   _printf
add     esp, 4
mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try block, setting
previous try level back to 0
$LN2@main:
mov     DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both blocks, setting
previous try level back to -2
jmp     SHORT $LN7@main

; outer block filter:
$LN8@main:
$LN19@main:
mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
mov     edx, DWORD PTR [ecx]
mov     eax, DWORD PTR [edx]
mov     DWORD PTR $T2[ebp], eax
mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
push   ecx
mov     edx, DWORD PTR $T2[ebp]
push   edx
call   _filter_user_exceptions
add     esp, 8
$LN10@main:
$LN17@main:
ret     0

; outer block handler:
$LN9@main:
mov     esp, DWORD PTR __$SEHRec$[ebp]
push   OFFSET $SG85503 ; 'user exception caught'
call   _printf
add     esp, 4
mov     DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both blocks, setting
previous try level back to -2
$LN7@main:
xor     eax, eax
mov     ecx, DWORD PTR __$SEHRec$[ebp+8]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

mov     DWORD PTR fs:0, ecx
pop     ecx
pop     edi
pop     esi
pop     ebx
mov     esp, ebp
pop     ebp
ret     0
_main   ENDP

_code$ = 8 ; size = 4
_ep$ = 12 ; size = 4
_filter_user_exceptions PROC
push    ebp
mov     ebp, esp
mov     eax, DWORD PTR _code$[ebp]
push    eax
push    OFFSET $SG85486 ; 'in filter. code=0x%08X'
call   _printf
add     esp, 8
cmp     DWORD PTR _code$[ebp], 1122867 ; 00112233H
jne     SHORT $LN2@filter_use
push    OFFSET $SG85488 ; 'yes, that is our exception'
call   _printf
add     esp, 4
mov     eax, 1
jmp     SHORT $LN3@filter_use
jmp     SHORT $LN3@filter_use
$LN2@filter_use:
push    OFFSET $SG85490 ; 'not our exception'
call   _printf
add     esp, 4
xor     eax, eax
$LN3@filter_use:
pop     ebp
ret     0
_filter_user_exceptions ENDP

```

Вот значение *cookies*: Cookie Offset это разница между адресом записанного в стеке значения EBP и значения  $EBP \oplus security\_cookie$  в стеке. Cookie XOR Offset это дополнительная разница между значением  $EBP \oplus security\_cookie$  и тем что записано в стеке. Если это уравнение не верно, то процесс остановится из-за разрушения стека:

$$security\_cookie \oplus (CookieXOROffset + address\_of\_saved\_EBP) == stack[address\_of\_saved\_EBP + CookieOffset]$$

Если Cookie Offset равно -2, это значит, что оно не присутствует.

## Windows x64

Как видно, это не самая быстрая штука, устанавливать SEH-структуры в каждом прологе функции. Еще одна проблема производительности — это менять

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

переменную *previous try level* много раз в течении исполнения функции. Так что в x64 всё сильно изменилось, теперь все указатели на try-блоки, функции фильтров и обработчиков, теперь записаны в другом PE-сегменте .pdata, откуда обработчик исключений **OS!** берет всю информацию.

Вот два примера из предыдущей секции, скомпилированных для x64:

Листинг 6.33: MSVC 2012

```

$SG86276 DB      'hello #1!', 0aH, 00H
$SG86277 DB      'hello #2!', 0aH, 00H
$SG86279 DB      'access violation, can''t recover', 0aH, 00H

pdata  SEGMENT
$pdata$main DD  imagerel $LN9
          DD      imagerel $LN9+61
          DD      imagerel $unwind$main
pdata  ENDS
pdata  SEGMENT
$pdata$main$filt$0 DD imagerel main$filt$0
          DD      imagerel main$filt$0+32
          DD      imagerel $unwind$main$filt$0
pdata  ENDS
xdata  SEGMENT
$unwind$main DD  020609H
          DD      030023206H
          DD      imagerel __C_specific_handler
          DD      01H
          DD      imagerel $LN9+8
          DD      imagerel $LN9+40
          DD      imagerel main$filt$0
          DD      imagerel $LN9+40
$unwind$main$filt$0 DD 020601H
          DD      050023206H
xdata  ENDS

_TEXT  SEGMENT
main   PROC
$LN9:
        push    rbx
        sub     rsp, 32
        xor     ebx, ebx
        lea    rcx, OFFSET FLAT:$SG86276 ; 'hello #1!'
        call   printf
        mov    DWORD PTR [rbx], 13
        lea    rcx, OFFSET FLAT:$SG86277 ; 'hello #2!'
        call   printf
        jmp    SHORT $LN8@main
$LN6@main:
        lea    rcx, OFFSET FLAT:$SG86279 ; 'access violation, can''t
recover'
        call   printf
        npad   1 ; align next label
$LN8@main:
        xor    eax, eax

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        add     rsp, 32
        pop     rbx
        ret     0
main     ENDP
_TEXT   ENDS

text$x   SEGMENT
main$filt$0 PROC
        push   rbp
        sub    rsp, 32
        mov    rbp, rdx
$LN5@main$filt$:
        mov    rax, QWORD PTR [rcx]
        xor    ecx, ecx
        cmp    DWORD PTR [rax], -1073741819; c0000005H
        sete   cl
        mov    eax, ecx
$LN7@main$filt$:
        add    rsp, 32
        pop    rbp
        ret    0
        int   3
main$filt$0 ENDP
text$x   ENDS

```

## Листинг 6.34: MSVC 2012

```

$SG86277 DB 'in filter. code=0x%08X', 0aH, 00H
$SG86279 DB 'yes, that is our exception', 0aH, 00H
$SG86281 DB 'not our exception', 0aH, 00H
$SG86288 DB 'hello!', 0aH, 00H
$SG86290 DB '0x112233 raised. now let''s crash', 0aH, 00H
$SG86292 DB 'access violation, can''t recover', 0aH, 00H
$SG86294 DB 'user exception caught', 0aH, 00H

pdata   SEGMENT
$pdata$filter_user_exceptions DD imagerel $LN6
        DD     imagerel $LN6+73
        DD     imagerel $unwind$filter_user_exceptions
$pdata$main DD imagerel $LN14
        DD     imagerel $LN14+95
        DD     imagerel $unwind$main
pdata   ENDS
pdata   SEGMENT
$pdata$main$filt$0 DD imagerel main$filt$0
        DD     imagerel main$filt$0+32
        DD     imagerel $unwind$main$filt$0
$pdata$main$filt$1 DD imagerel main$filt$1
        DD     imagerel main$filt$1+30
        DD     imagerel $unwind$main$filt$1
pdata   ENDS

xdata   SEGMENT
$unwind$filter_user_exceptions DD 020601H

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        DD      030023206H
$unwind$main DD 020609H
        DD      030023206H
        DD      imagerel __C_specific_handler
        DD      02H
        DD      imagerel $LN14+8
        DD      imagerel $LN14+59
        DD      imagerel main$filt$0
        DD      imagerel $LN14+59
        DD      imagerel $LN14+8
        DD      imagerel $LN14+74
        DD      imagerel main$filt$1
        DD      imagerel $LN14+74
$unwind$main$filt$0 DD 020601H
        DD      050023206H
$unwind$main$filt$1 DD 020601H
        DD      050023206H
xdata  ENDS

_TEXT  SEGMENT
main   PROC
$LN14:
        push   rbx
        sub    rsp, 32
        xor    ebx, ebx
        lea   rcx, OFFSET FLAT:$SG86288 ; 'hello!'
        call  printf
        xor   r9d, r9d
        xor   r8d, r8d
        xor   edx, edx
        mov   ecx, 1122867 ; 00112233H
        call  QWORD PTR __imp_RaiseException
        lea  rcx, OFFSET FLAT:$SG86290 ; '0x112233 raised. now let''s
crash' call  printf
        mov   DWORD PTR [rbx], 13
        jmp   SHORT $LN13@main
$LN11@main:
        lea  rcx, OFFSET FLAT:$SG86292 ; 'access violation, can''t
recover' call  printf
        npad 1 ; align next label
$LN13@main:
        jmp   SHORT $LN9@main
$LN7@main:
        lea  rcx, OFFSET FLAT:$SG86294 ; 'user exception caught'
        call  printf
        npad 1 ; align next label
$LN9@main:
        xor   eax, eax
        add   rsp, 32
        pop   rbx
        ret   0
main   ENDP

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

text$x SEGMENT
main$filt$0 PROC
    push    rbp
    sub     rsp, 32
    mov     rbp, rdx
$LN10@main$filt$:
    mov     rax, QWORD PTR [rcx]
    xor     ecx, ecx
    cmp     DWORD PTR [rax], -1073741819; c0000005H
    sete   cl
    mov     eax, ecx
$LN12@main$filt$:
    add     rsp, 32
    pop     rbp
    ret     0
    int     3
main$filt$0 ENDP

main$filt$1 PROC
    push    rbp
    sub     rsp, 32
    mov     rbp, rdx
$LN6@main$filt$:
    mov     rax, QWORD PTR [rcx]
    mov     rdx, rcx
    mov     ecx, DWORD PTR [rax]
    call    filter_user_exceptions
    npad   1 ; align next label
$LN8@main$filt$:
    add     rsp, 32
    pop     rbp
    ret     0
    int     3
main$filt$1 ENDP
text$x ENDS

_TEXT SEGMENT
code$ = 48
ep$ = 56
filter_user_exceptions PROC
$LN6:
    push    rbx
    sub     rsp, 32
    mov     ebx, ecx
    mov     edx, ecx
    lea    rcx, OFFSET FLAT:$SG86277 ; 'in filter. code=0x%08X'
    call   printf
    cmp    ebx, 1122867; 00112233H
    jne    SHORT $LN2@filter_use
    lea    rcx, OFFSET FLAT:$SG86279 ; 'yes, that is our exception'
    call   printf
    mov    eax, 1

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        add    rsp, 32
        pop    rbx
        ret    0
$LN2@filter_use:
        lea    rcx, OFFSET FLAT:$SG86281 ; 'not our exception'
        call   printf
        xor    eax, eax
        add    rsp, 32
        pop    rbx
        ret    0
filter_user_exceptions ENDP
_TEXT    ENDS

```

Смотрите [Igor Skochinsky, *Compiler Internals: Exceptions and RTTI*, (2012)]<sup>48</sup> для более детального описания.

Помимо информации об исключениях, секция `.pdata` также содержит начала и концы почти всех функций, так что эту информацию можно использовать в каких-либо утилитах, предназначенных для автоматизации анализа.

### Больше о SEH

[Matt Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Handling*, (1997)]<sup>49</sup>, [Igor Skochinsky, *Compiler Internals: Exceptions and RTTI*, (2012)]<sup>50</sup>.

## 6.5.4. Windows NT: Критические секции

Критические секции в любой **OS!** очень важны в мультитредовой среде, используются в основном для обеспечения гарантии что только один тред будет иметь доступ к данным в один момент времени, блокируя остальные треды и прерывания.

Вот как структура `CRITICAL_SECTION` объявлена в линейке OS [Windows NT](#):

Листинг 6.35: (Windows Research Kernel v1.2) `public/sdk/inc/nturtl.h`

```

typedef struct _RTL_CRITICAL_SECTION {
    PRTL_CRITICAL_SECTION_DEBUG DebugInfo;

    //
    // The following three fields control entering and exiting the critical
    // section for the resource
    //
    LONG LockCount;
    LONG RecursionCount;
    HANDLE OwningThread;           // from the thread's ClientId->UniqueThread

```

<sup>48</sup>Также доступно здесь: <http://yurichev.com/mirrors/RE/Recon-2012-Skochinsky-Compiler-Internals.pdf>

<sup>49</sup>Также доступно здесь: <http://www.microsoft.com/msj/0197/Exception/Exception.aspx>

<sup>50</sup>Также доступно здесь: <http://yurichev.com/mirrors/RE/Recon-2012-Skochinsky-Compiler-Internals.pdf>



```

HANDLE LockSemaphore;
ULONG_PTR SpinCount;           // force size on 64-bit systems when packed
} RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;

```

Вот как работает функция EnterCriticalSection():

Листинг 6.36: Windows 2008/ntdll.dll/x86 (begin)

```

_RtlEnterCriticalSection@4
var_C          = dword ptr -0Ch
var_8          = dword ptr -8
var_4          = dword ptr -4
arg_0         = dword ptr  8

        mov     edi, edi
        push   ebp
        mov     ebp, esp
        sub     esp, 0Ch
        push   esi
        push   edi
        mov     edi, [ebp+arg_0]
        lea     esi, [edi+4] ; LockCount
        mov     eax, esi
        lock btr dword ptr [eax], 0
        jnb     wait ; jump if CF=0

loc_7DE922DD:
        mov     eax, large fs:18h
        mov     ecx, [eax+24h]
        mov     [edi+0Ch], ecx
        mov     dword ptr [edi+8], 1
        pop     edi
        xor     eax, eax
        pop     esi
        mov     esp, ebp
        pop     ebp
        retn   4

... skipped

```

Самая важная инструкция в этом фрагменте кода — это BTR (с префиксом LOCK): нулевой бит сохраняется в флаге CF и очищается в памяти. Это **атомарная операция**, блокирующая доступ всех остальных процессоров к этому значению в памяти (обратите внимание на префикс LOCK перед инструкцией BTR).

Если бит в LockCount является 1, хорошо, сбросить его и вернуться из функции: мы в критической секции. Если нет — критическая секция уже занята другим тредом, тогда ждем.

Ожидание там сделано через вызов WaitForSingleObject().

А вот как работает функция LeaveCriticalSection():

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Листинг 6.37: Windows 2008/ntdll.dll/x86 (begin)

```

_RtlLeaveCriticalSection@4 proc near
arg_0          = dword ptr 8

                mov     edi, edi
                push   ebp
                mov     ebp, esp
                push   esi
                mov     esi, [ebp+arg_0]
                add     dword ptr [esi+8], 0FFFFFFFFh ; RecursionCount
                jnz     short loc_7DE922B2
                push   ebx
                push   edi
                lea     edi, [esi+4] ; LockCount
                mov     dword ptr [esi+0Ch], 0
                mov     ebx, 1
                mov     eax, edi
                lock xadd [eax], ebx
                inc     ebx
                cmp     ebx, 0FFFFFFFFh
                jnz     loc_7DEA8EB7

loc_7DE922B0:
                pop     edi
                pop     ebx

loc_7DE922B2:
                xor     eax, eax
                pop     esi
                pop     ebp
                retn    4

... skipped

```

XADD это «обменять и прибавить». В данном случае, это значит прибавить 1 к значению в LockCount, при этом сохранить изначальное значение LockCount в регистре EBX. Впрочем, значение в EBX позже инкрементируется при помощи последующей инструкции INC EBX, и оно также будет равно обновленному значению LockCount.

Эта операция также атомарная, потому что также имеет префикс LOCK, что означает, что другие CPU или ядра CPU в системе не будут иметь доступа к этой ячейке памяти .

Префикс LOCK очень важен: два треда, каждый из которых работает на разных CPU или ядрах CPU, могут попытаться одновременно войти в критическую секцию, одновременно модифицируя значение в памяти, и это может привести к непредсказуемым результатам.

## Глава 7

# Инструменты

### 7.1. Дизассемблеры

#### 7.1.1. IDA

Старая бесплатная версия доступна для скачивания <sup>1</sup>.

Краткий справочник горячих клавиш: ?? (стр. ??)

### 7.2. Отладчики

#### 7.2.1. OllyDbg

Очень популярный отладчик пользовательской среды win32: [ollydbg.de](http://ollydbg.de).

Краткий справочник горячих клавиш: ?? (стр. ??)

#### 7.2.2. GDB

Не очень популярный отладчик у реверсеров, тем не менее, крайне удобный.

Некоторые команды: ?? (стр. ??).

#### 7.2.3. tracer

Автор часто использует *tracer* <sup>2</sup> вместо отладчика.

Со временем, автор этих строк отказался использовать отладчик, потому что всё что ему нужно от него это иногда подсмотреть какие-либо аргументы какой-либо функции во время исполнения или состояние регистров в определенном

---

<sup>1</sup>[hex-rays.com/products/ida/support/download\\_freeware.shtml](http://hex-rays.com/products/ida/support/download_freeware.shtml)

<sup>2</sup>[yurichev.com](http://yurichev.com)

месте. Каждый раз загружать отладчик для этого это слишком, поэтому родилась очень простая утилита *tracer*. Она консольная, запускается из командной строки, позволяет перехватывать исполнение функций, ставить точки останова на произвольные места, смотреть состояние регистров, модифицировать их, и т. д.

Но для учебы очень полезно трассировать код руками в отладчике, наблюдать как меняются значения регистров (например, как минимум классический SoftICE, OllyDbg, WinDbg подсвечивают измененные регистры), флагов, данные, менять их самому, смотреть реакцию, и т. д.

## 7.3. Трассировка системных вызовов

### strace / dtruss

Позволяет показать, какие системные вызовы (syscalls(?? (стр. ??))) прямо сейчас вызывает процесс.

Например:

```
# strace df -h
...
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or ↵
↳ directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF↵
↳ \1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220\232\1\0004\0\0\0"... , ↵
↳ 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1770984, ...}) = 0
mmap2(NULL, 1780508, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) ↵
↳ = 0xb75b3000
```

В Mac OS X для этого же имеется *dtruss*.

В Cygwin также есть *strace*, впрочем, насколько известно, он показывает результаты только для .exe-файлов скомпилированных для среды самого cygwin.

## 7.4. Декомпиляторы

Пока существует только один публично доступный декомпилятор в Си высокого качества: Hex-Rays: [hex-rays.com/products/decompiler/](http://hex-rays.com/products/decompiler/)

Читайте больше о нем: ?? (стр. ??).

## 7.5. Прочие инструменты

- Microsoft Visual Studio Express<sup>3</sup>: Усеченная бесплатная версия Visual Studio, пригодная для простых экспериментов. Некоторые полезные опции: ?? (стр. ??).
- Hiew<sup>4</sup>: для мелкой модификации кода в исполняемых файлах.
- binary grep: небольшая утилита для поиска констант (либо просто последовательности байт) в большом количестве файлов, включая неисполняемые: [GitHub](#). В rada.re имеется также rafind2 для тех же целей.

### 7.5.1. Калькуляторы

Хороший калькулятор для нужд реверс-инженера должен поддерживать как минимум десятичную, шестнадцатеричную и двоичную системы счисления, а также многие важные операции как “исключающее ИЛИ” и сдвиги.

- В IDA есть встроенный калькулятор (“?”).
- В rada.re есть *rax2*.
- <https://yurichev.com/progcalc/>
- Стандартный калькулятор в Windows имеет режим *программистского калькулятора*.

## 7.6. Чего-то здесь не хватает?

Если вы знаете о хорошем инструменте, которого не хватает здесь в этом списке, пожалуйста сообщите мне об этом:

[мой адреса](#).

---

<sup>3</sup>[visualstudio.com/en-US/products/visual-studio-express-vs](https://visualstudio.com/en-US/products/visual-studio-express-vs)

<sup>4</sup>[hiew.ru](http://hiew.ru)

## Глава 8

# Примеры из практики

Вместо эпиграфа:

**Питер Сейбел:** Как вы читаете исходный код? Ведь непросто читать даже то, что написано на известном вам языке программирования.

**Дональд Кнут:** Но это действительно того стоит, если говорить о том, что выстраивается в вашей голове. Как я читаю код? Когда-то была машина под названием Bunker Ramo 300, и кто-то мне однажды сказал, что компилятор Фортрана для этой машины работает чрезвычайно быстро, но никто не понимает почему. Я получил копию его исходного кода. У меня не было руководства по этому компьютеру, поэтому я даже не был уверен, какой это был машинный язык.

Но я взялся за это, посчитав интересной задачей. Я нашел BEGIN и начал разбираться. В кодах операций есть ряд двухбуквенных мнемоник, поэтому я мог начать анализировать: “Возможно, это инструкция загрузки, а это, возможно, инструкция перехода”. Кроме того, я знал, что это компилятор Фортрана, и иногда он обращался к седьмой колонке перфокарты - там он мог определить, комментарий это или нет.

Спустя три часа я кое-что понял об этом компьютере. Затем обнаружил огромные таблицы ветвлений. То есть это была своего рода головоломка, и я продолжал рисовать небольшие схемы, как разведчик, пытающийся разгадать секретный шифр. Но я знал, что программа работает, и знал, что это компилятор Фортрана — это не был шифр, в том смысле что программа не была написана с сознательной целью запутать. Все дело было в коде, поскольку у меня не было руководства по компьютеру.

В конце концов мне удалось выяснить, почему компилятор работал так быстро. К сожалению, дело было не в гениальных алгоритмах - просто там применялись методы неструктурированного

программирования и код был максимально оптимизирован вручную.

По большому счету, именно так и должна решаться головоломка: составляются таблицы, схемы, информация извлекается по крупицам, выдвигается гипотеза. В общем, когда я читаю техническую работу, это такая же сложная задача. Я пытаюсь влезть в голову автора, понять, в чем состоял его замысел. Чем больше вы учитесь читать вещи, написанные другими, тем более способны изобретать что-то свое - так мне кажется.

( Сейбел Питер — Кодеры за работой. Размышления о ремесле программиста )

## 8.1. Шутка с Маджонгом (Windows 7)

Маджонг это замечательная игра, однако, можем ли мы усложнить её, запретив пункт меню *Hint* (подсказка)?

В Windows 7, я могу найти файлы Mahjong.dll и Mahjong.exe в:

```
C:\Windows\winsxs\
x86_microsoft-windows-s..inboxgames-shanghai_31bf3856ad364e35_6.1.7600.16385_none\
c07a51d9507d9398.
```

Также, файл Mahjong.exe.mui в:

```
C:\Windows\winsxs\
x86_microsoft-windows-s..-shanghai.resources_31bf3856ad364e35_6.1.7600.16385_en-us
_c430954533c66bf3
```

и в

```
C:\Windows\winsxs\
x86_microsoft-windows-s..-shanghai.resources_31bf3856ad364e35_6.1.7600.16385_ru-ru
_0d51acf984cb679a.
```

Я использую англоязычную Windows, но с поддержкой русского языка, так что тут могут быть файлы ресурсов для двух языков. Открыв файл Mahjong.exe.mui в Resource Hacker, можно увидеть определения меню:

Листинг 8.1: Ресурсы меню в Mahjong.exe.mui

```
103 MENU
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
{
  POPUP "&Game"
  {
    MENUITEM "&New Game\tF2", 40000
    MENUITEM SEPARATOR
    MENUITEM "&Undo\tCtrl+Z", 40001
    MENUITEM "&Hint\tH", 40002
    MENUITEM SEPARATOR
    MENUITEM "&Statistics\tF4", 40003
    MENUITEM "&Options\tF5", 40004
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

MENUITEM "Change &Appearance\tF7", 40005
MENUITEM SEPARATOR
MENUITEM "E&xit", 40006
}
POPUP "&Help"
{
MENUITEM "&View Help\tF1", 40015
MENUITEM "&About Mahjong Titans", 40016
MENUITEM SEPARATOR
MENUITEM "Get &More Games Online", 40020
}
}

```

Подменю *Hint* имеет код 40002. Открываю Mahjong.exe в IDA и ищу значение 40002.

(Я пишу это в ноябре 2019. Почему-то, IDA не может выкачать PDB-файлы с серверов Microsoft. Может быть, Windows 7 больше не поддерживается? Так или иначе, имен ф-ций я не смог установить...)

Листинг 8.2: Mahjong.exe

```

.text:010205C8 6A 03          push 3
.text:010205CA 85 FF          test edi, edi
.text:010205CC 5B             pop ebx

...

.text:01020625 57             push edi ; uIDEnableItem
.text:01020626 FF 35 C8 97 08 01 push hmenu ; hMenu
.text:0102062C FF D6          call esi ; EnableMenuItem
.text:0102062E 83 7D 08 01    cmp [ebp+arg_0], 1
.text:01020632 BF 42 9C 00 00  mov edi, 40002
.text:01020637 75 18          jnz short loc_1020651 ; переход должен
    быть всегда
.text:01020639 6A 00          push 0 ; uEnable
.text:0102063B 57             push edi ; uIDEnableItem
.text:0102063C FF 35 B4 8B 08 01 push hMenu ; hMenu
.text:01020642 FF D6          call esi ; EnableMenuItem
.text:01020644 6A 00          push 0 ; uEnable
.text:01020646 57             push edi ; uIDEnableItem
.text:01020647 FF 35 C8 97 08 01 push hmenu ; hMenu
.text:0102064D FF D6          call esi ; EnableMenuItem
.text:0102064F EB 1A          jmp short loc_102066B
.text:01020651
.text:01020651          loc_1020651: ; CODE XREF: sub_1020581+B6
.text:01020651 53             push ebx ; 3
.text:01020652 57             push edi ; uIDEnableItem
.text:01020653 FF 35 B4 8B 08 01 push hMenu ; hMenu
.text:01020659 FF D6          call esi ; EnableMenuItem
.text:0102065B 53             push ebx ; 3
.text:0102065C 57             push edi ; uIDEnableItem
.text:0102065D FF 35 C8 97 08 01 push hmenu ; hMenu
.text:01020663 FF D6          call esi ; EnableMenuItem

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



Эта часть кода разрешает или запрещает пункт меню *Hint*.

И согласно MSDN<sup>1</sup>:

MF\_DISABLED | MF\_GRAYED = 3 и MF\_ENABLED = 0.

Я думаю, эта ф-ция разрешает или запрещает несколько пунктов меню (*Hint*, *Undo*, и т. д.), исходя из значения в `arg_0`. Потому что при старте, когда пользователь выбирает тип игры, пункты *Hint* и *Undo* запрещены. Они разрешены, когда игра начинается.

Так что я изменяю в файле `Mahjong.exe` по адресу `0x01020637` байт `0x75` на `0xEB`, делая так, что переход `JNZ` всегда будет работать. Эффект в том, что ф-ция всегда будет вызываться как `EnableMenuItem(..., ..., 3)`. Теперь подменю *Hint* всё время запрещено.

Также, почему-то, ф-ция `EnableMenuItem()` вызывается дважды, для `hMenu` и для `hmenu`. Может быть, в программе два меню, и может быть, они переключаются?

В качестве домашней работы, попробуйте запретить подменю *Undo*, чтобы сделать игру еще труднее.

## 8.2. Шутка с task manager (Windows Vista)

Посмотрим, сможем ли мы немного хакнуть Task Manager, чтобы он находил больше ядер в **CPU!**, чем присутствует.

В начале задумаемся, откуда Task Manager знает количество ядер?

В `win32` имеется функция `GetSystemInfo()`, при помощи которой можно узнать.

Но она не импортируется в `taskmgr.exe`. Есть еще одна в `NTAPI`, `NtQuerySystemInformation()`, которая используется в `taskmgr.exe` в ряде мест.

Чтобы узнать количество ядер, нужно вызвать эту функцию с константной `SystemBasicInformation` в первом аргументе (а это ноль

<sup>2</sup>).

Второй аргумент должен указывать на буфер, который примет всю информацию.

Так что нам нужно найти все вызовы функции `NtQuerySystemInformation(0, ?, ?, ?)`.

Откроем `taskmgr.exe` в IDA. Что всегда хорошо с исполняемыми файлами от Microsoft, это то что IDA может скачать соответствующий `PDB`-файл именно для этого файла и добавить все имена функций.

Видимо, Task Manager написан на Си++ и некоторые функции и классы имеют говорящие за себя имена.

---

<sup>1</sup><https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-enablemenuitem>

<sup>2</sup>MSDN

Тут есть классы CAdapter, CNetPage, CPerfPage, CProcInfo, CProcPage, CSvcPage, CTaskPage, CUserPage. Должно быть, каждый класс соответствует каждой вкладке в Task Manager.

Пройдемся по всем вызовам и добавим комментарий с числом, передающимся как первый аргумент.

В некоторых местах напишем «not zero», потому что значение в тех местах однозначно не ноль, но что-то другое (больше об этом во второй части главы).

А мы все-таки ищем ноль передаваемый как аргумент.

Dire...	T.	Address	Text	
U	Up	p	wWinMain+50E	call cs:__imp_NtQuerySystemInformation; 0
U	Up	p	wWinMain+542	call cs:__imp_NtQuerySystemInformation; 2
U	Up	p	CPerfPage::TimerEvent(void)+200	call cs:__imp_NtQuerySystemInformation; not zero
U		p	InitPerfInfo(void)+2C	call cs:__imp_NtQuerySystemInformation; 0
U	D...	p	InitPerfInfo(void)+F0	call cs:__imp_NtQuerySystemInformation; 8
U	D...	p	CalcCpuTime(int)+5F	call cs:__imp_NtQuerySystemInformation; 8
U	D...	p	CalcCpuTime(int)+248	call cs:__imp_NtQuerySystemInformation; 2
U	D...	p	CPerfPage::CalcPhysicalMem(unsigned ...	call cs:__imp_NtQuerySystemInformation; not zero
U	D...	p	CPerfPage::CalcPhysicalMem(unsigned ...	call cs:__imp_NtQuerySystemInformation; not zero
U	D...	p	CProcPage::GetProcessInfo(void)+2B	call cs:__imp_NtQuerySystemInformation; 5
U	D...	p	CProcPage::UpdateProcInfoArray(void)+...	call cs:__imp_NtQuerySystemInformation; 0
U	D...	p	CProcPage::UpdateProcInfoArray(void)+...	call cs:__imp_NtQuerySystemInformation; 2
U	D...	p	CProcPage::Initialize(HWND__ *)+201	call cs:__imp_NtQuerySystemInformation; 0
U	D...	p	CProcPage::GetTaskListEx(void)+3C	call cs:__imp_NtQuerySystemInformation; 5

Рис. 8.1: IDA: вызовы функции NtQuerySystemInformation()

Да, имена действительно говорящие сами за себя.

Когда мы внимательно изучим каждое место, где вызывается NtQuerySystemInformation(0, ?, ?, ?), то быстро найдем то что нужно в функции InitPerfInfo():

Листинг 8.3: taskmgr.exe (Windows Vista)

```
.text:10000B4B3      xor     r9d, r9d
.text:10000B4B6      lea    rdx, [rsp+0C78h+var_C58] ; buffer
.text:10000B4BB      xor     ecx, ecx
.text:10000B4BD      lea    ebp, [r9+40h]
.text:10000B4C1      mov    r8d, ebp
.text:10000B4C4      call   cs:__imp_NtQuerySystemInformation ; 0
.text:10000B4CA      xor     ebx, ebx
.text:10000B4CC      cmp    eax, ebx
.text:10000B4CE      jge    short loc_10000B4D7
.text:10000B4D0
.text:10000B4D0 loc_10000B4D0: ; CODE XREF:
InitPerfInfo(void)+97
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:1000B4D0                                     ;
        InitPerfInfo(void)+AF
.text:1000B4D0                                     xor     al, al
.text:1000B4D2                                     jmp     loc_1000B5EA
.text:1000B4D7 ;
-----
.text:1000B4D7                                     ; CODE XREF:
.text:1000B4D7 loc_1000B4D7:
        InitPerfInfo(void)+36
.text:1000B4D7         mov     eax, [rsp+0C78h+var_C50]
.text:1000B4DB         mov     esi, ebx
.text:1000B4DD         mov     r12d, 3E80h
.text:1000B4E3         mov     cs:?g_PageSize@@3KA, eax ; ulong g_PageSize
.text:1000B4E9         shr     eax, 0Ah
.text:1000B4EC         lea    r13, __ImageBase
.text:1000B4F3         imul   eax, [rsp+0C78h+var_C4C]
.text:1000B4F8         cmp    [rsp+0C78h+var_C20], bpl
.text:1000B4FD         mov    cs:?g_MEMMax@@3_JA, rax ; __int64 g_MEMMax
.text:1000B504         movzx  eax, [rsp+0C78h+var_C20] ; number of CPUs
.text:1000B509         cmova  eax, ebp
.text:1000B50C         cmp    al, bl
.text:1000B50E         mov    cs:?g_cProcessors@@3EA, al ;
        uchar g_cProcessors

```

`g_cProcessors` это глобальная переменная и это имя присвоено IDA в соответствии с PDB-файлом, скачанным с сервера символов Microsoft.

Байт берется из `var_C20`. И `var_C58` передается в `NtQuerySystemInformation()` как указатель на принимающий буфер. Разница между `0xC20` и `0xC58` это `0x38` (56). Посмотрим на формат структуры, который можно найти в MSDN:

```

typedef struct _SYSTEM_BASIC_INFORMATION {
    BYTE Reserved1[24];
    PVOID Reserved2[4];
    CCHAR NumberOfProcessors;
} SYSTEM_BASIC_INFORMATION;

```

Это система x64, так что каждый `PVOID` занимает здесь 8 байт.

Так что все *reserved*-поля занимают  $24 + 4 * 8 = 56$ .

О да, это значит, что `var_C20` в локальном стеке это именно поле `NumberOfProcessors` структуры `SYSTEM_BASIC_INFORMATION`.

Проверим нашу догадку. Скопируем `taskmgr.exe` из `C:\Windows\System32` в какую-нибудь другую папку (чтобы *Windows Resource Protection* не пыталась восстанавливать измененный `taskmgr.exe`).

Откроем его в `Hiew` и найдем это место:

```

01 0000B4F8: 40386C2458      cmp     [rsp][058],bp1
01 0000B4FD: 48890544A00100  mov     [00000001`00025548],rax
01 0000B504: 0F66442458      movzx  eax,b,[rsp][058]
01 0000B509: 0F47C5          cmova  eax,ebp
01 0000B50C: 3AC3           cmp     al,b1
01 0000B50E: 880574950100    mov     [00000001`00024A88],al
01 0000B514: 7645           jbe    .00000001`0000B55B --03
01 0000B516: 488BFB        mov     rdi,rbx
01 0000B519: 498BD4        mov     rdx,r12
01 0000B51C: 8BCD          mov     ecx,ebp

```

Рис. 8.2: Hiew: найдем это место

Заменяем инструкцию MOVZX на нашу.

Сделаем вид что у нас 64 ядра процессора. Добавим дополнительную инструкцию **NOP!** (потому что наша инструкция короче чем та что там сейчас):

```

00 0000A8F8: 40386C2458      cmp     [rsp][058],bp1
00 0000A8FD: 48890544A00100  mov     [000024948],rax
00 0000A904: 66B84000        mov     ax,00040 ; '@'
00 0000A908: 90              nop
00 0000A909: 0F47C5          cmova  eax,ebp
00 0000A90C: 3AC3           cmp     al,b1
00 0000A90E: 880574950100    mov     [000023E88],al
00 0000A914: 7645           jbe    00000A95B
00 0000A916: 488BFB        mov     rdi,rbx
00 0000A919: 498BD4        mov     rdx,r12
00 0000A91C: 8BCD          mov     ecx,ebp

```

Рис. 8.3: Hiew: меняем инструкцию

И это работает! Конечно же, данные в графиках неправильные. Иногда, Task Manager даже показывает общую загрузку CPU более 100%.

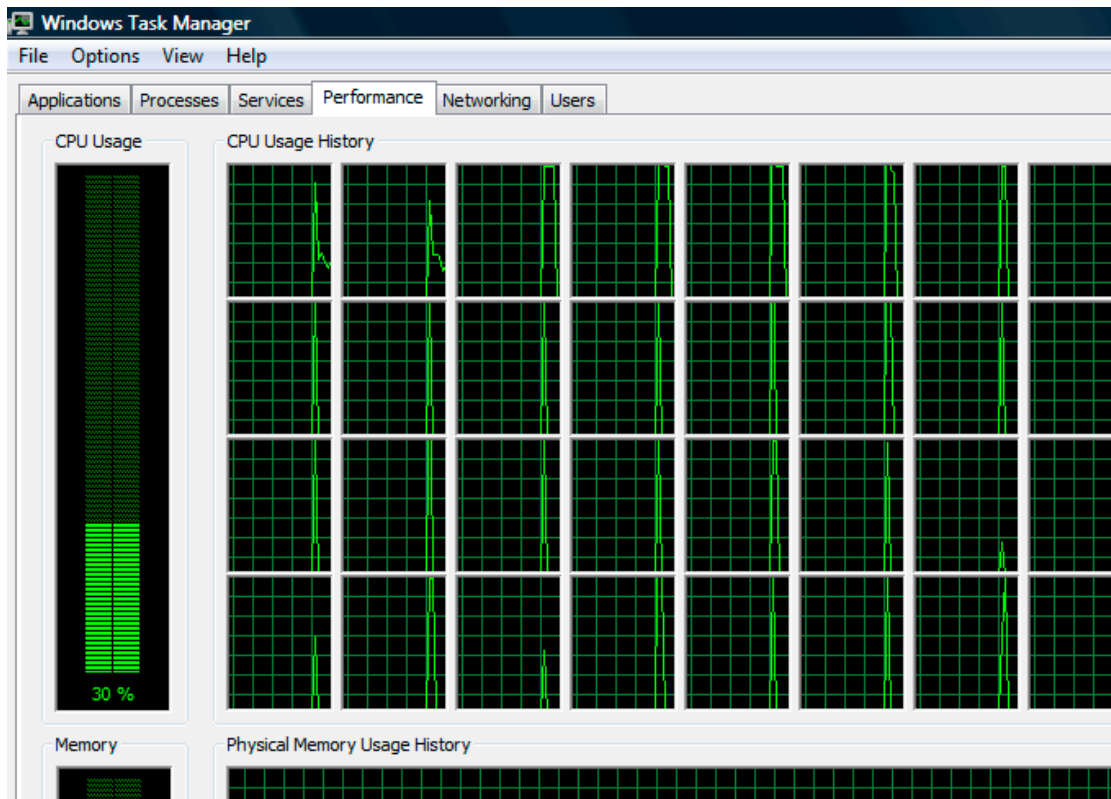


Рис. 8.4: Обманутый Windows Task Manager

Самое большое число, при котором Task Manager не падает, это 64.

Должно быть, Task Manager в Windows Vista не тестировался на компьютерах с большим количеством ядер.

И, наверное, там есть внутри какие-то статичные структуры данных, ограниченные до 64-х ядер.

### 8.2.1. Использование LEA для загрузки значений

Иногда, LEA используется в taskmgr.exe вместо MOV для установки первого аргумента

NtQuerySystemInformation():

Листинг 8.4: taskmgr.exe (Windows Vista)

```

xor     r9d, r9d
div     dword ptr [rsp+4C8h+WndClass.lpfWndProc]
lea     rdx, [rsp+4C8h+VersionInformation]
lea     ecx, [r9+2]      ; put 2 to ECX
mov     r8d, 138h
mov     ebx, eax

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; ECX=SystemPerformanceInformation
    call    cs:__imp_NtQuerySystemInformation ; 2

    ...

    mov     r8d, 30h
    lea    r9, [rsp+298h+var_268]
    lea    rdx, [rsp+298h+var_258]
    lea    ecx, [r8-2Dh] ; put 3 to ECX
; ECX=SystemTimeOfDayInformation
    call    cs:__imp_NtQuerySystemInformation ; not zero

    ...

    mov     rbp, [rsi+8]
    mov     r8d, 20h
    lea    r9, [rsp+98h+arg_0]
    lea    rdx, [rsp+98h+var_78]
    lea    ecx, [r8+2Fh] ; put 0x4F to ECX
    mov     [rsp+98h+var_60], ebx
    mov     [rsp+98h+var_68], rbp
; ECX=SystemSuperfetchInformation
    call    cs:__imp_NtQuerySystemInformation ; not zero

```

Должно быть, **MSVC!** сделал так, потому что код инструкции LEA короче чем MOV REG, 5 (было бы 5 байт вместо 4).

LEA со смещением в пределах -128..127 (смещение будет занимать 1 байт в опкоде) с 32-битными регистрами даже еще короче (из-за отсутствия REX-префикса) — 3 байта.

Еще один пример подобного: ?? (стр. ??).

## 8.3. Шутка с игрой Color Lines

Это очень популярная игра с большим количеством реализаций. Возьмем одну из них, с названием BallTriX, от 1997, доступную бесплатно на [https://archive.org/details/BallTriX\\_1020](https://archive.org/details/BallTriX_1020)<sup>3</sup>. Вот как она выглядит:

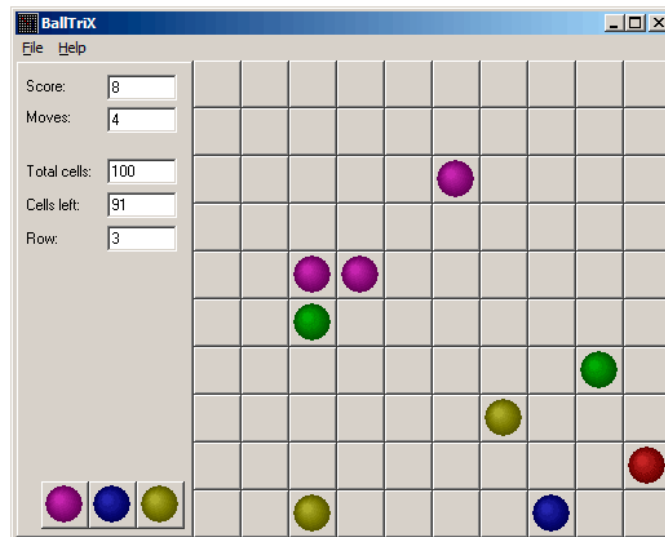


Рис. 8.5: Обычный вид игры

---

<sup>3</sup>Или на <https://web.archive.org/web/20141110053442/http://www.download-central.ws/Win32/Games/B/BallTriX/> или <http://www.benya.com/balltrix/>.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Посмотрим, сможем ли мы найти генератор псевдослучайных чисел и сделать с ним одну шутку.

**IDA!** быстро распознает стандартную функцию `_rand` в `balltrix.exe` по адресу `0x00403DA0`. **IDA!** также показывает, что она вызывается только из одного места:

```
.text:00402C9C sub_402C9C      proc near          ;
                CODE XREF: sub_402ACA+52
.text:00402C9C                                     ; sub_402ACA+64 ...
.text:00402C9C
.text:00402C9C arg_0          = dword ptr 8
.text:00402C9C
.text:00402C9C         push     ebp
.text:00402C9D         mov      ebp, esp
.text:00402C9F         push     ebx
.text:00402CA0         push     esi
.text:00402CA1         push     edi
.text:00402CA2         mov      eax, dword_40D430
.text:00402CA7         imul    eax, dword_40D440
.text:00402CAE         add      eax, dword_40D5C8
.text:00402CB4         mov      ecx, 32000
.text:00402CB9         cdq
.text:00402CBA         idiv    ecx
.text:00402CBC         mov      dword_40D440, edx
.text:00402CC2         call    _rand
.text:00402CC7         cdq
.text:00402CC8         idiv    [ebp+arg_0]
.text:00402CCB         mov      dword_40D430, edx
.text:00402CD1         mov      eax, dword_40D430
.text:00402CD6         jmp     $+5
.text:00402CDB         pop      edi
.text:00402CDC         pop      esi
.text:00402CDD         pop      ebx
.text:00402CDE         leave
.text:00402CDF         retn
.text:00402CDF sub_402C9C      endp
```

Назовем её «random». Пока не будем концентрироваться на самом коде функции.

Эта функция вызывается из трех мест.

Вот первые два:

```
.text:00402B16         mov      eax, dword_40C03C ; 10 here
.text:00402B1B         push    eax
.text:00402B1C         call    random
.text:00402B21         add     esp, 4
.text:00402B24         inc     eax
.text:00402B25         mov     [ebp+var_C], eax
.text:00402B28         mov     eax, dword_40C040 ; 10 here
.text:00402B2D         push    eax
.text:00402B2E         call    random
.text:00402B33         add     esp, 4
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



Вот третье:

```
.text:00402BBB      mov     eax, dword_40C058 ; 5 here
.text:00402BC0      push   eax
.text:00402BC1      call   random
.text:00402BC6      add    esp, 4
.text:00402BC9      inc    eax
```

Так что у функции только один аргумент. 10 передается в первых двух случаях и 5 в третьем.

Мы также можем заметить, что размер доски 10\*10 и здесь 5 возможных цветов. Это оно! Стандартная функция `rand()` возвращает число в пределах  $0..0x7FFF$  и это неудобно, так что многие программисты пишут свою функцию, возвращающую случайное число в некоторых заданных пределах. В нашем случае, предел это  $0..n-1$  и  $n$  передается как единственный аргумент в функцию. Мы можем быстро проверить это в отладчике.

Сделаем так, чтобы третий вызов функции всегда возвращал ноль. В начале заменим три инструкции (PUSH/CALL/ADD) на **NOP**!s. Затем добавим инструкцию `XOR EAX, EAX`, для очистки регистра EAX.

```
.00402BB8: 83C410      add     esp, 010
.00402BBB: A158C04000  mov     eax, [00040C058]
.00402BC0: 31C0       xor     eax, eax
.00402BC2: 90        nop
.00402BC3: 90        nop
.00402BC4: 90        nop
.00402BC5: 90        nop
.00402BC6: 90        nop
.00402BC7: 90        nop
.00402BC8: 90        nop
.00402BC9: 40        inc    eax
.00402BCA: 8B4DF8     mov     ecx, [ebp] [-8]
.00402BCD: 8D0C49     lea    ecx, [ecx] [ecx]*2
.00402BD0: 8B15F4D54000  mov     edx, [00040D5F4]
```

Что мы сделали, это заменили вызов функции `random()` на код, всегда возвращающий ноль.

Теперь запустим:

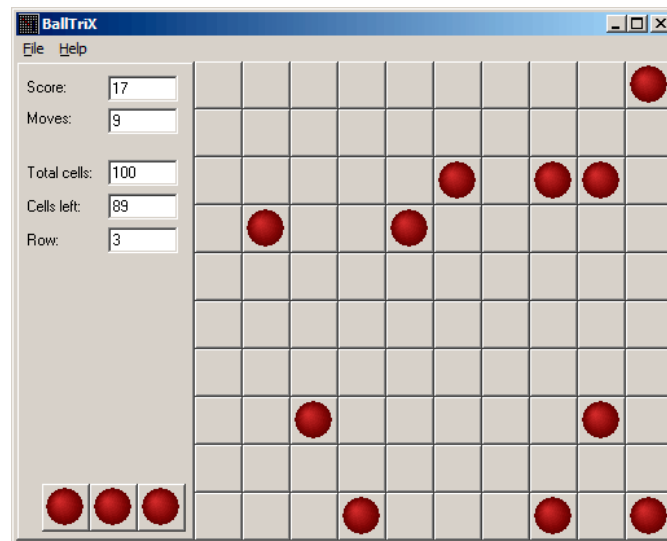


Рис. 8.6: Шутка сработала

О да, это работает<sup>4</sup>.

Но почему аргументы функции `random()` это глобальные переменные? Это просто потому что в настройках игры можно изменять размер доски, так что эти параметры не фиксированы. 10 и 5 это просто значения по умолчанию.

## 8.4. Сапёр (Windows XP)

Для тех, кто не очень хорошо играет в Сапёра (Minesweeper), можно попробовать найти все скрытые мины в отладчике.

Как мы знаем, Сапёр располагает мины случайным образом, так что там должен быть генератор случайных чисел или вызов стандартной функции `Si rand()`.

Вот что хорошо в реверсинге продуктов от Microsoft, так это то что часто есть **PDB**-файл со всеми символами (имена функций, и т.д.).

Когда мы загружаем `winmine.exe` в **IDA!**, она скачивает **PDB** файл именно для этого исполняемого файла и добавляет все имена.

И вот оно, только один вызов `rand()` в этой функции:

```
.text:01003940 ; __stdcall Rnd(x)
.text:01003940 _Rnd@4      proc near          ; CODE XREF:
                StartGame()+53
.text:01003940          ; StartGame()+61
.text:01003940
```

<sup>4</sup>Автор этой книги однажды сделал это как шутку для его сотрудников, в надежде что они перестанут играть. Надежды не оправдались.

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:01003940 arg_0          = dword ptr 4
.text:01003940
.text:01003940          call   ds:__imp__rand
.text:01003946          cdq
.text:01003947          idiv  [esp+arg_0]
.text:0100394B          mov   eax, edx
.text:0100394D          retn  4
.text:0100394D _Rnd@4          endp

```

Так её назвала **IDA!** и это было имя данное ей разработчиками Сапёра.

Функция очень простая:

```

int Rnd(int limit)
{
    return rand() % limit;
};

```

(В **PDB**-файле не было имени «limit»; это мы назвали этот аргумент так, вручную.)

Так что она возвращает случайное число в пределах от нуля до заданного предела.

Rnd() вызывается только из одного места, это функция с названием StartGame(), и как видно, это именно тот код, что расставляет мины:

```

.text:010036C7          push  _xBoxMac
.text:010036CD          call  _Rnd@4           ; Rnd(x)
.text:010036D2          push  _yBoxMac
.text:010036D8          mov   esi, eax
.text:010036DA          inc   esi
.text:010036DB          call  _Rnd@4           ; Rnd(x)
.text:010036E0          inc   eax
.text:010036E1          mov   ecx, eax
.text:010036E3          shl  ecx, 5           ; ECX=ECX*32
.text:010036E6          test  _rgBlk[ecx+esi], 80h
.text:010036EE          jnz  short loc_10036C7
.text:010036F0          shl  eax, 5           ; EAX=EAX*32
.text:010036F3          lea  eax, _rgBlk[eax+esi]
.text:010036FA          or   byte ptr [eax], 80h
.text:010036FD          dec  _cBombStart
.text:01003703          jnz  short loc_10036C7

```

Сапёр позволяет задать размеры доски, так что X (xBoxMac) и Y (yBoxMac) это глобальные переменные.

Они передаются в Rnd() и генерируются случайные координаты. Мина устанавливается инструкцией OR на 0x010036FA. И если она уже была установлена до этого (это возможно, если пара функций Rnd() сгенерирует пару, которая уже была сгенерирована), тогда TEST и JNZ на 0x010036E6 перейдет на повторную генерацию пары.

cBombStart это глобальная переменная, содержащая количество мин. Так что это цикл.

Ширина двумерного массива это 32 (мы можем это вывести, глядя на инструкцию SHL, которая умножает одну из координат на 32).

Размер глобального массива rgBlk можно легко узнать по разнице между меткой rgBlk в сегменте данных и следующей известной меткой. Это 0x360 (864):

.data:01005340 _rgBlk	db 360h dup(?)	; DATA XREF:
MainWndProc(x,x,x,x)+574		
.data:01005340		; DisplayBlk(x,x)+23
.data:010056A0 _Preferences	dd ?	; DATA XREF:
FixMenus()+2		
...		

$864/32 = 27$ .

Так что размер массива  $27 * 32$ ? Это близко к тому что мы знаем: если попытаемся установить размер доски в установках Сапёра на  $100 * 100$ , то он установит размер  $24 * 30$ . Так что это максимальный размер доски здесь. И размер массива фиксирован для доски любого размера.

Посмотрим на всё это в OllyDbg. Запустим Сапёр, присоединим (attach) OllyDbg к нему и увидим содержимое памяти по адресу где массив rgBlk (0x01005340) <sup>5</sup>.

Так что у нас выходит такой дамп памяти массива:

Address	Hex dump
01005340	10 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F
01005350	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
01005360	10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005370	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
01005380	10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005390	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
010053A0	10 0F 0F 0F 0F 0F 0F 0F 8F 0F 10 0F 0F 0F 0F 0F
010053B0	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
010053C0	10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
010053D0	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
010053E0	10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
010053F0	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
01005400	10 0F 0F 8F 0F 0F 8F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005410	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
01005420	10 8F 0F 0F 8F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005430	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
01005440	10 8F 0F 0F 0F 0F 8F 0F 0F 8F 10 0F 0F 0F 0F 0F
01005450	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
01005460	10 0F 0F 0F 0F 8F 0F 0F 0F 8F 10 0F 0F 0F 0F 0F
01005470	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
01005480	10 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F
01005490	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F

<sup>5</sup>Все адреса здесь для Сапёра под Windows XP SP3 English. Они могут отличаться для других сервис-паков.

```

010054A0  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
010054B0  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
010054C0  0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|

```

OlyDbg, как и любой другой шестнадцатеричный редактор, показывает 16 байт на строку. Так что каждая 32-байтная строка массива занимает ровно 2 строки.

Это уровень для начинающих (доска 9\*9).

Тут еще какая-то квадратная структура, заметная визуально (байты 0x10).

Нажмем «Run» в OlyDbg чтобы разморозить процесс Сапёра, потом нажмем в случайное место окна Сапёра, попадаем на mine, но теперь видны все мины:



Рис. 8.7: Мины

Сравнивая места с минами и дампы, мы можем обнаружить что 0x10 это граница, 0x0F — пустой блок, 0x8F — мина. Вероятно 0x10 это т.н., *sentinel value*.

Теперь добавим комментарии и также заключим все байты 0x8F в квадратные скобки:

```

border:
01005340  10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F
01005350  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #1:
01005360  10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005370  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #2:
01005380  10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005390  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #3:
010053A0  10 0F 0F 0F 0F 0F 0F 0F[8F]0F 10 0F 0F 0F 0F 0F
010053B0  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #4:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

010053C0 10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
010053D0 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #5:
010053E0 10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
010053F0 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #6:
01005400 10 0F 0F[8F]0F 0F[8F]0F 0F 0F 10 0F 0F 0F 0F 0F
01005410 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #7:
01005420 10[8F]0F 0F[8F]0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005430 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #8:
01005440 10[8F]0F 0F 0F 0F[8F]0F 0F[8F]10 0F 0F 0F 0F 0F
01005450 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #9:
01005460 10 0F 0F 0F 0F[8F]0F 0F 0F[8F]10 0F 0F 0F 0F 0F
01005470 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
border:
01005480 10 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F
01005490 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F

```

Теперь уберем все байты связанные с границами (0x10) и всё что за ними:

```

0F 0F 0F 0F 0F 0F 0F 0F
0F 0F 0F 0F 0F 0F 0F 0F
0F 0F 0F 0F 0F 0F 0F[8F]0F
0F 0F 0F 0F 0F 0F 0F 0F
0F 0F 0F 0F 0F 0F 0F 0F
0F 0F[8F]0F 0F[8F]0F 0F 0F
[8F]0F 0F[8F]0F 0F 0F 0F 0F
[8F]0F 0F 0F 0F[8F]0F 0F[8F]
0F 0F 0F 0F[8F]0F 0F 0F[8F]

```

Да, это всё мины, теперь это очень хорошо видно, в сравнении со скриншотом.

Вот что интересно, это то что мы можем модифицировать массив прямо в OllyDbg.

Уберем все мины заменив все байты 0x8F на 0x0F, и вот что получится в Сапёре:

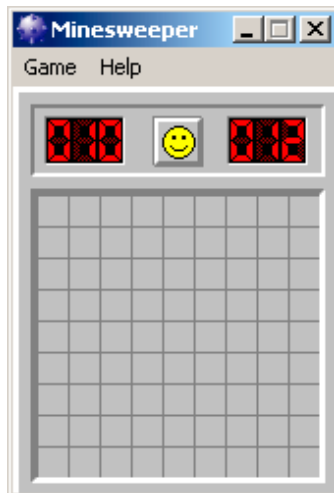


Рис. 8.8: Все мины убраны в отладчике

Также уберем их все и добавим их в первом ряду:

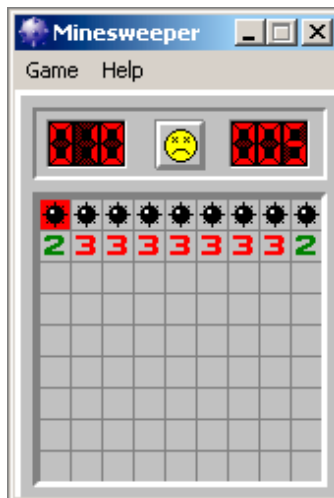


Рис. 8.9: Мины, установленные в отладчике

Отладчик не очень удобен для подсматривания (а это была наша изначальная цель), так что напишем маленькую утилиту для показа содержимого доски:

```
// Windows XP MineSweeper cheater
// written by dennis(a)yurichev.com for http://beginners.re/ book
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

#include <windows.h>
#include <assert.h>
#include <stdio.h>

int main (int argc, char * argv[])
{
    int i, j;
    HANDLE h;
    DWORD PID, address, rd;
    BYTE board[27][32];

    if (argc!=3)
    {
        printf ("Usage: %s <PID> <address>\n", argv[0]);
        return 0;
    };

    assert (argv[1]!=NULL);
    assert (argv[2]!=NULL);

    assert (sscanf (argv[1], "%d", &PID)==1);
    assert (sscanf (argv[2], "%x", &address)==1);

    h=OpenProcess (PROCESS_VM_OPERATION | PROCESS_VM_READ | ↵
    ↵ PROCESS_VM_WRITE, FALSE, PID);

    if (h==NULL)
    {
        DWORD e=GetLastError();
        printf ("OpenProcess error: %08X\n", e);
        return 0;
    };

    if (ReadProcessMemory (h, (LPVOID)address, board, sizeof(board), &↵
    ↵ rd)!=TRUE)
    {
        printf ("ReadProcessMemory() failed\n");
        return 0;
    };

    for (i=1; i<26; i++)
    {
        if (board[i][0]==0x10 && board[i][1]==0x10)
            break; // end of board
        for (j=1; j<31; j++)
        {
            if (board[i][j]==0x10)
                break; // board border
            if (board[i][j]==0x8F)
                printf ("*");
            else
                printf (" ");
        }
    }
}

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```
};  
    printf ("\n");  
};  
  
    CloseHandle (h);  
};
```

Просто установите **PID!**<sup>6 7</sup> и адрес массива (0x01005340 для Windows XP SP3 English) и она покажет его <sup>8</sup>.

Она подключается к win32-процессу по **PID!**-у и просто читает из памяти процесса по этому адресу.

### 8.4.1. Автоматический поиск массива

Задавать адрес каждый раз при запуске нашей утилиты, это неудобно. К тому же, разные версии "Сапёра" могут иметь этот массив по разным адресам. Зная, что всегда есть рамка (байты 0x10), массив легко найти в памяти:

```
// find frame to determine the address  
process_mem=(BYTE*)malloc(process_mem_size);  
assert (process_mem!=NULL);  
  
if (ReadProcessMemory (h, (LPVOID)start_addr, process_mem, ↵  
↳ process_mem_size, &rd)!=TRUE)  
{  
    printf ("ReadProcessMemory() failed\n");  
    return 0;  
};  
  
// for 9*9 grid.  
// FIXME: slow!  
for (i=0; i<process_mem_size; i++)  
{  
    if (memcmp(process_mem+i, "\x10\x10\x10\x10\x10\x10\x10\x10 ↵  
↳ \x10\x10\x10\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F ↵  
↳ x0F\x0F\x0F\x0F\x0F\x0F\x0F\x10", 32)==0)  
    {  
        // found  
        address=start_addr+i;  
        break;  
    };  
};  
if (address==0)  
{  
    printf ("Can't determine address of frame (and grid)\n");  
    return 0;  
}  
else
```

<sup>6</sup>**PID!**

<sup>7</sup>PID можно увидеть в Task Manager (это можно включить в «View → Select Columns»)

<sup>8</sup>Скомпилированная версия здесь: [beginners.re](http://beginners.re)

```
{  
    printf ("Found frame and grid at 0x%x\n", address);  
};
```

Полный исходный код: [https://beginners.re/paywall/RE4B-source/current-tree/examples/minesweeper/minesweeper\\_cheater2.c](https://beginners.re/paywall/RE4B-source/current-tree/examples/minesweeper/minesweeper_cheater2.c).

### 8.4.2. Упражнения

- Почему байты описывающие границы (0x10) (или *sentinel value*) присутствуют вообще? Зачем они нужны, если они вообще не видимы в интерфейсе Сапёра? Как можно обойтись без них?
- Как выясняется, здесь больше возможных значений (для открытых блоков, для тех на которых игрок установил флажок, и т.д.).  
Попробуйте найти значение каждого.
- Измените мою утилиту так, чтобы она в запущенном процессе Сапёра убирала все мины, или расставляла их в соответствии с каким-то заданным шаблоном.

## 8.5. Хакаем часы в Windows

Иногда я устраиваю первоапрельские пранки для моих сотрудников.

Посмотрим, можем ли мы сделать что-то с часами в Windows? Можем ли мы их заставить идти в обратную сторону?

Прежде всего, когда вы кликаете на часы/время в строке состояния (*status bar*), запускается модуль `C:\WINDOWS\SYSTEM32\TIMEDATE.CPL`, а это обычный **PE!**-файл.

Посмотрим, как отрисовываются стрелки? Когда я открываю этот файл (из Windows 7) в Resource Hacker, здесь есть разные виды циферблата, но нет стрелок:

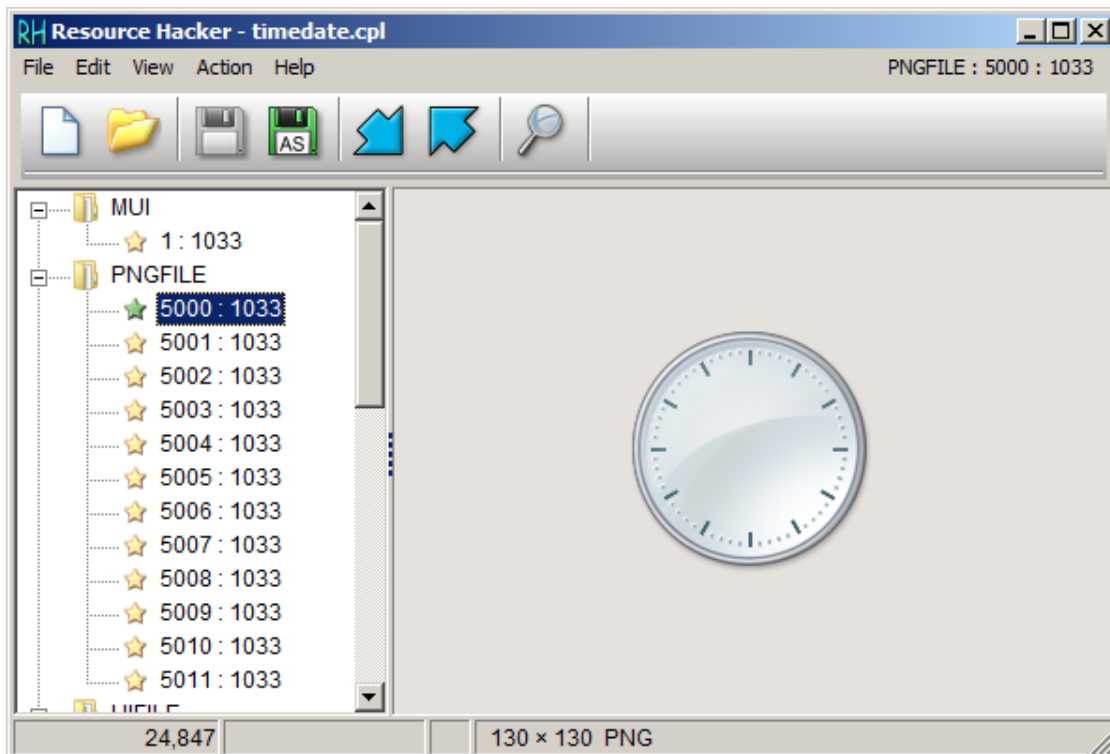


Рис. 8.10: Resource Hacker

ОК, что мы знаем? Как рисовать стрелку часов? Они все начинаются в середине круга и заканчиваются на его границе. Следовательно, нам нужно рассчитать координаты точки на границе круга. Из школьной математики мы можем вспомнить, что для рисования круга нужно использовать  $\phi$ -ции синуса/косинуса, или хотя бы квадратного корня. Такого в *TIMEDATE.CPL* нет, по крайней мере на первый взгляд. Но, благодаря отладочным PDB-файлам от Microsoft, я могу найти  $\phi$ -цию с названием *CAnalogClock::DrawHand()*, которая вызывает *Gdiplus::Graphics::DrawLine()* минимум дважды.

Вот её код:

```
.text:6EB9DBC7 ; private: enum Gdiplus::Status __thiscall
    CAnalogClock::_DrawHand(class Gdiplus::Graphics *, int, struct ClockHand
    const &, class Gdiplus::Pen *)
.text:6EB9DBC7 ?_DrawHand@CAnalogClock@@AAE?@
    ↪ AW4Status@Gdiplus@@PAVGraphics@3@HABUClockHand@@PAVPen@3@@Z proc near
.text:6EB9DBC7 ; CODE XREF: CAnalogClock::_ClockPaint(HDC__ *)+163
.text:6EB9DBC7 ; CAnalogClock::_ClockPaint(HDC__ *)+18B
.text:6EB9DBC7
.text:6EB9DBC7 var_10 = dword ptr -10h
.text:6EB9DBC7 var_C = dword ptr -0Ch
.text:6EB9DBC7 var_8 = dword ptr -8
.text:6EB9DBC7 var_4 = dword ptr -4
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:6EB9DBC7 arg_0      = dword ptr 8
.text:6EB9DBC7 arg_4      = dword ptr 0Ch
.text:6EB9DBC7 arg_8      = dword ptr 10h
.text:6EB9DBC7 arg_C      = dword ptr 14h
.text:6EB9DBC7
.text:6EB9DBC7          mov     edi, edi
.text:6EB9DBC9          push   ebp
.text:6EB9DBCA          mov     ebp, esp
.text:6EB9DBCC          sub     esp, 10h
.text:6EB9DBCF          mov     eax, [ebp+arg_4]
.text:6EB9DBD2          push   ebx
.text:6EB9DBD3          push   esi
.text:6EB9DBD4          push   edi
.text:6EB9DBD5          cdq
.text:6EB9DBD6          push   3Ch
.text:6EB9DBD8          mov     esi, ecx
.text:6EB9DBDA          pop     ecx
.text:6EB9DBDB          idiv   ecx
.text:6EB9DBDD          push   2
.text:6EB9DBDF          lea    ebx, table[edx*8]
.text:6EB9DBE6          lea    eax, [edx+1Eh]
.text:6EB9DBE9          cdq
.text:6EB9DBEA          idiv   ecx
.text:6EB9DBEC          mov     ecx, [ebp+arg_0]
.text:6EB9DBEF          mov     [ebp+var_4], ebx
.text:6EB9DBF2          lea    eax, table[edx*8]
.text:6EB9DBF9          mov     [ebp+arg_4], eax
.text:6EB9DBFC          call   ?
      ↓
      ↓ SetInterpolationMode@Graphics@Gdiplus@@QAE?
      ↓ Aw4Status@2@W4InterpolationMode@2@@Z ;
      Gdiplus::Graphics::SetInterpolationMode(Gdiplus::InterpolationMode)
.text:6EB9DC01          mov     eax, [esi+70h]
.text:6EB9DC04          mov     edi, [ebp+arg_8]
.text:6EB9DC07          mov     [ebp+var_10], eax
.text:6EB9DC0A          mov     eax, [esi+74h]
.text:6EB9DC0D          mov     [ebp+var_C], eax
.text:6EB9DC10          mov     eax, [edi]
.text:6EB9DC12          sub     eax, [edi+8]
.text:6EB9DC15          push   8000          ; nDenominator
.text:6EB9DC1A          push   eax          ; nNumerator
.text:6EB9DC1B          push   dword ptr [ebx+4] ; nNumber
.text:6EB9DC1E          mov     ebx, ds:__imp__MulDiv@12 ;
      MulDiv(x,x,x)
.text:6EB9DC24          call   ebx ; MulDiv(x,x,x) ; MulDiv(x,x,x)
.text:6EB9DC26          add     eax, [esi+74h]
.text:6EB9DC29          push   8000          ; nDenominator
.text:6EB9DC2E          mov     [ebp+arg_8], eax
.text:6EB9DC31          mov     eax, [edi]
.text:6EB9DC33          sub     eax, [edi+8]
.text:6EB9DC36          push   eax          ; nNumerator
.text:6EB9DC37          mov     eax, [ebp+var_4]
.text:6EB9DC3A          push   dword ptr [eax] ; nNumber
.text:6EB9DC3C          call   ebx ; MulDiv(x,x,x) ; MulDiv(x,x,x)
.text:6EB9DC3E          add     eax, [esi+70h]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:6EB9DC41      mov     ecx, [ebp+arg_0]
.text:6EB9DC44      mov     [ebp+var_8], eax
.text:6EB9DC47      mov     eax, [ebp+arg_8]
.text:6EB9DC4A      mov     [ebp+var_4], eax
.text:6EB9DC4D      lea    eax, [ebp+var_8]
.text:6EB9DC50      push   eax
.text:6EB9DC51      lea    eax, [ebp+var_10]
.text:6EB9DC54      push   eax
.text:6EB9DC55      push   [ebp+arg_C]
.text:6EB9DC58      call   ?DrawLine@Graphics@Gdiplus@@QAE?@
      ↓ Aw4Status@2@PBVPen@2@ABVPoint@2@1@Z ;
      Gdiplus::Graphics::DrawLine(Gdiplus::Pen const *,Gdiplus::Point const
      &,Gdiplus::Point const &)
.text:6EB9DC5D      mov     ecx, [edi+8]
.text:6EB9DC60      test   ecx, ecx
.text:6EB9DC62      jbe    short loc_6EB9DCAA
.text:6EB9DC64      test   eax, eax
.text:6EB9DC66      jnz    short loc_6EB9DCAA
.text:6EB9DC68      mov     eax, [ebp+arg_4]
.text:6EB9DC6B      push   8000             ; nDenominator
.text:6EB9DC70      push   ecx             ; nNumerator
.text:6EB9DC71      push   dword ptr [eax+4] ; nNumber
.text:6EB9DC74      call   ebx ; MulDiv(x,x,x) ; MulDiv(x,x,x)
.text:6EB9DC76      add    eax, [esi+74h]
.text:6EB9DC79      push   8000             ; nDenominator
.text:6EB9DC7E      push   dword ptr [edi+8] ; nNumerator
.text:6EB9DC81      mov     [ebp+arg_8], eax
.text:6EB9DC84      mov     eax, [ebp+arg_4]
.text:6EB9DC87      push   dword ptr [eax] ; nNumber
.text:6EB9DC89      call   ebx ; MulDiv(x,x,x) ; MulDiv(x,x,x)
.text:6EB9DC8B      add    eax, [esi+70h]
.text:6EB9DC8E      mov     ecx, [ebp+arg_0]
.text:6EB9DC91      mov     [ebp+var_8], eax
.text:6EB9DC94      mov     eax, [ebp+arg_8]
.text:6EB9DC97      mov     [ebp+var_4], eax
.text:6EB9DC9A      lea    eax, [ebp+var_8]
.text:6EB9DC9D      push   eax
.text:6EB9DC9E      lea    eax, [ebp+var_10]
.text:6EB9DCA1      push   eax
.text:6EB9DCA2      push   [ebp+arg_C]
.text:6EB9DCA5      call   ?DrawLine@Graphics@Gdiplus@@QAE?@
      ↓ Aw4Status@2@PBVPen@2@ABVPoint@2@1@Z ;
      Gdiplus::Graphics::DrawLine(Gdiplus::Pen const *,Gdiplus::Point const
      &,Gdiplus::Point const &)
.text:6EB9DCAA
.text:6EB9DCAA loc_6EB9DCAA: ; CODE XREF:
      CAnalogClock::_DrawHand(Gdiplus::Graphics *,int,ClockHand const
      &,Gdiplus::Pen *)+9B
.text:6EB9DCAA      ; CAnalogClock::_DrawHand(Gdiplus::Graphics
      *,int,ClockHand const &,Gdiplus::Pen *)+9F
.text:6EB9DCAA      pop    edi
.text:6EB9DCAB      pop    esi
.text:6EB9DCAC      pop    ebx
.text:6EB9DCAD      leave
.text:6EB9DCAE      retn  10h

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
.text:6EB9DCAE ?_DrawHand@CAnalogClock@@AAE?↵
      ↵ Aw4Status@Gdiplus@@PAVGraphics@3@HABUClockHand@@PAVPen@3@@Z endp
.text:6EB9DCAE
```

Мы можем увидеть что аргументы *DrawLine()* зависят от результата ф-ции *MulDiv()* и таблицы *table[]* (название дал я), которая содержит 8-байтные элементы (посмотрите на второй операнд LEA).

Что внутри *table[]*?

```
.text:6EB87890 ; int table[]
.text:6EB87890 table          dd 0
.text:6EB87894              dd 0FFFFFF0C1h
.text:6EB87898              dd 344h
.text:6EB8789C              dd 0FFFFFF0ECh
.text:6EB878A0              dd 67Fh
.text:6EB878A4              dd 0FFFFFF16Fh
.text:6EB878A8              dd 9A8h
.text:6EB878AC              dd 0FFFFFF248h
.text:6EB878B0              dd 0CB5h
.text:6EB878B4              dd 0FFFFFF374h
.text:6EB878B8              dd 0F9Fh
.text:6EB878BC              dd 0FFFFFF4F0h
.text:6EB878C0              dd 125Eh
.text:6EB878C4              dd 0FFFFFF6B8h
.text:6EB878C8              dd 14E9h
```

...

Доступ к ней есть только из ф-ции *DrawHand()*. У ней 120 32-битных слов или 60 32-битных пар ...подождите, 60? Посмотрим ближе на эти значения. Прежде всего, я затру нулями первые 6 пар или 12 32-битных слов, и затем я положу пропатченный *TIMEDATE.CPL* в *C:\WINDOWS\SYSTEM32*. (Вам, возможно, придется установить владельца файла *\*TIMEDATE.CPL\** равным вашей первичной пользовательской учетной записи (вместо *TrustedInstaller*), а также загрузиться в безопасном режиме с командной строкой, чтобы скопировать этот файл, который обычно залоченный.)

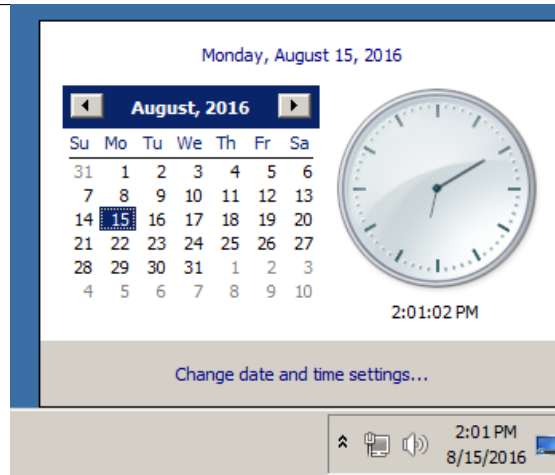


Рис. 8.11: Attempt to run

Теперь, когда стрелка находится на 0..5 секундах/минутах, она невидимая! Хотя, противоположная (короткая) часть секундной стрелки видима, и движется. Когда любая стрелка за пределами этой области, она видима, как обычно.

Посмотрим на эту таблицу при помощи Mathematica. Я скопировал таблицу из *TIMEDATE.CPL* в файл *tbl* (480 байт). Будем считать, что это знаковые значения, потому что половина элементов меньше нуля (0FFFFE0C1h, и т. д.). Если бы эти значения были бы беззнаковыми, они были бы подозрительно большими.

```
In[]:= tbl = BinaryReadList["~/.../tbl", "Integer32"]
Out[]= {0, -7999, 836, -7956, 1663, -7825, 2472, -7608, 3253, -7308, 3999, ↵
↵ ↵
-6928, 4702, -6472, 5353, -5945, 5945, -5353, 6472, -4702, 6928, \
-4000, 7308, -3253, 7608, -2472, 7825, -1663, 7956, -836, 8000, 0, \
7956, 836, 7825, 1663, 7608, 2472, 7308, 3253, 6928, 4000, 6472, \
4702, 5945, 5353, 5353, 5945, 4702, 6472, 3999, 6928, 3253, 7308, \
2472, 7608, 1663, 7825, 836, 7956, 0, 7999, -836, 7956, -1663, 7825, \
-2472, 7608, -3253, 7308, -4000, 6928, -4702, 6472, -5353, 5945, \
-5945, 5353, -6472, 4702, -6928, 3999, -7308, 3253, -7608, 2472, \
-7825, 1663, -7956, 836, -7999, 0, -7956, -836, -7825, -1663, -7608, \
-2472, -7308, -3253, -6928, -4000, -6472, -4702, -5945, -5353, -5353, \
-5945, -4702, -6472, -3999, -6928, -3253, -7308, -2472, -7608, -1663, \
-7825, -836, -7956}

In[]:= Length[tbl]
Out[]= 120
```

Будем считать два последовательно идущих 32-битных значения как пару:

```
In[]:= pairs = Partition[tbl, 2]
Out[]= {{0, -7999}, {836, -7956}, {1663, -7825}, {2472, -7608}, \
{3253, -7308}, {3999, -6928}, {4702, -6472}, {5353, -5945}, {5945, \
-5353}, {6472, -4702}, {6928, -4000}, {7308, -3253}, {7608, -2472}, \
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
{7825, -1663}, {7956, -836}, {8000, 0}, {7956, 836}, {7825,
1663}, {7608, 2472}, {7308, 3253}, {6928, 4000}, {6472,
4702}, {5945, 5353}, {5353, 5945}, {4702, 6472}, {3999,
6928}, {3253, 7308}, {2472, 7608}, {1663, 7825}, {836, 7956}, {0,
7999}, {-836, 7956}, {-1663, 7825}, {-2472, 7608}, {-3253,
7308}, {-4000, 6928}, {-4702, 6472}, {-5353, 5945}, {-5945,
5353}, {-6472, 4702}, {-6928, 3999}, {-7308, 3253}, {-7608,
2472}, {-7825, 1663}, {-7956, 836}, {-7999,
0}, {-7956, -836}, {-7825, -1663}, {-7608, -2472}, {-7308, -3253}, \
{-6928, -4000}, {-6472, -4702}, {-5945, -5353}, {-5353, -5945}, \
{-4702, -6472}, {-3999, -6928}, {-3253, -7308}, {-2472, -7608}, \
{-1663, -7825}, {-836, -7956}}
```

```
In[]:= Length[pairs]
Out[]= 60
```

Попробуем считать каждую пару как координату X/Y и нарисуем все 60 пар, а также первые 15 пар:



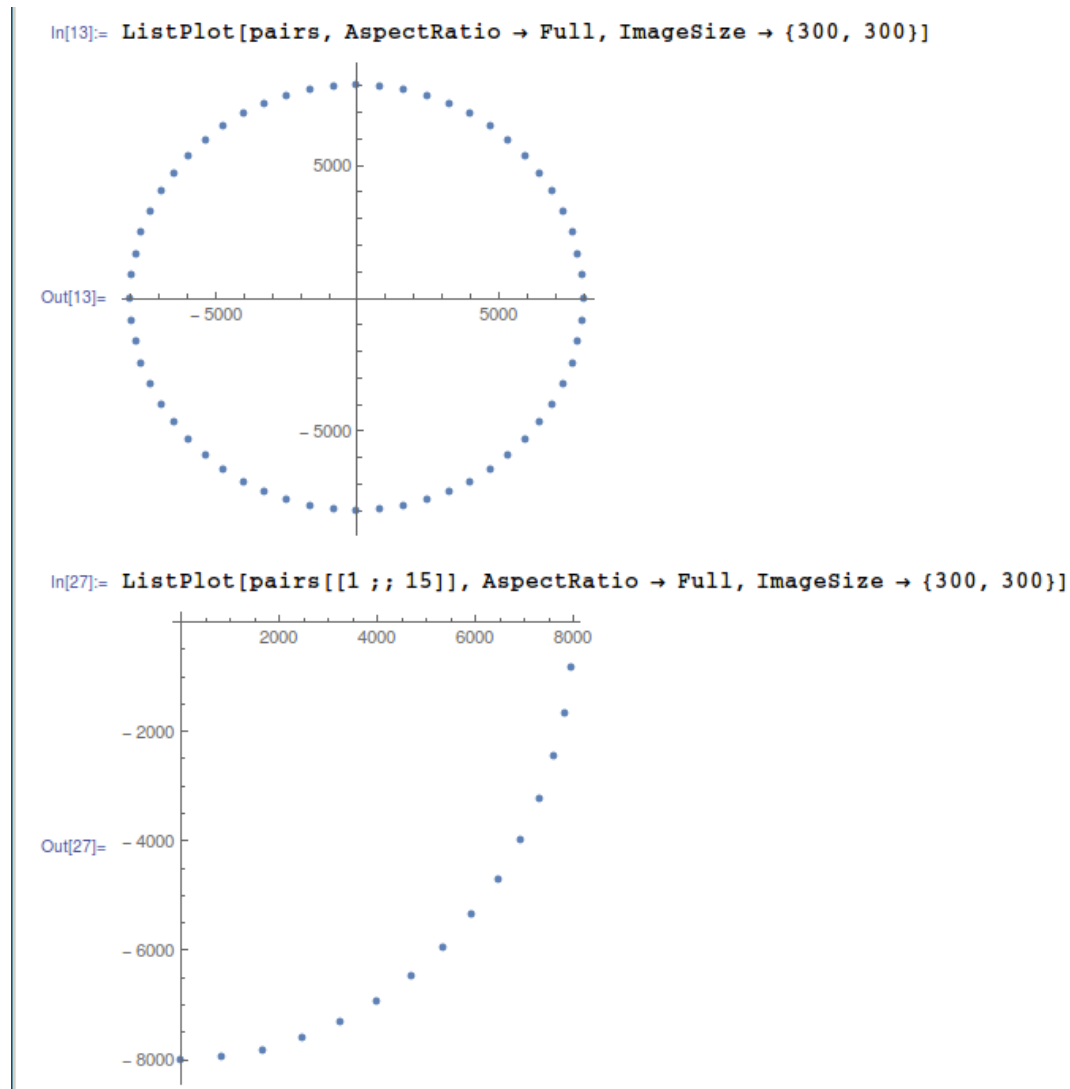


Рис. 8.12: Mathematica

Ну теперь это кое-что! Каждая пара это просто координата. Первые 15 пар это координаты  $\frac{1}{4}$  круга.

Видимо, разработчики в Microsoft рассчитали все координаты предварительно и сохранили их в таблице. Это распространенная, хотя и немного олд-скульная практика – доступ к предвычисленным значениям в таблице быстрее, чем вызывать относительно медленные ф-ции синуса/косинуса<sup>9</sup>. В наше время операции синуса/косинуса уже не такие *дорогие*.

Теперь понятно, почему когда я затер первые 6 пар, стрелки были невидимы

<sup>9</sup>Сегодня это называют *memoization*

в этой области: на самом деле, стрелки рисовались, просто их длина была нулевой, потому что стрелка начиналась в координатах 0:0, и там же и заканчивалась.

### Пранк (шутка)

Учитывая всё это, можем ли мы заставить стрелки идти в обратную сторону? На самом деле, это просто, нужно просто развернуть таблицу, так что каждая стрелка, вместо отображения на месте нулевой секунды, рисовалась бы на месте 59-й секунды.

Когда-то давным давно я сделал патчер, в в самом начале 2000-х, для Windows 2000. Трудно поверить, но он всё еще работает и для Windows 7, видимо, таблица с тех пор не менялась!

Исходник патчера: [https://beginners.re/paywall/RE4B-source/current-tree/examples/timedate/time\\_pt.c](https://beginners.re/paywall/RE4B-source/current-tree/examples/timedate/time_pt.c).

Теперь видно как стрелки идут назад:

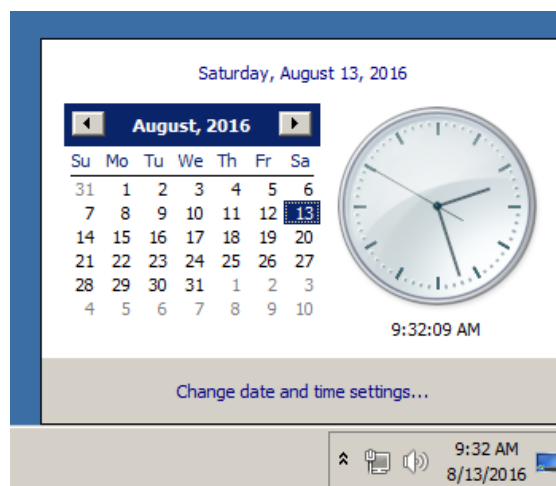


Рис. 8.13: Now it works

В этой книге, конечно, нет анимации, но если присмотритесь, увидите, что на самом деле стрелки показывают корректное время, но весь циферблат повернут вертикально, как если бы мы видели его изнутри часов.

### Утекшие исходники Windows 2000

Так что я сделал патчер, потом утекли исходники Windows 2000 (я не могу заставить вас поверить мне, конечно). Посмотрим на исходный код этой ф-ции и таблицы.

Нужный файл это `win2k/private/shell/cpls/utc/clock.c`:

```
//
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

// Array containing the sine and cosine values for hand positions.
//
POINT rCircleTable[] =
{
    { 0,      -7999},
    { 836,   -7956},
    { 1663,  -7825},
    { 2472,  -7608},
    { 3253,  -7308},
    ...
    { -4702, -6472},
    { -3999, -6928},
    { -3253, -7308},
    { -2472, -7608},
    { -1663, -7825},
    { -836 , -7956},
};

/////////////////////////////////////////////////////////////////
//
// DrawHand
//
// Draws the hands of the clock.
//
/////////////////////////////////////////////////////////////////

void DrawHand(
    HDC hDC,
    int pos,
    HPEN hPen,
    int scale,
    int patMode,
    PLOCKSTR np)
{
    LPPPOINT lppt;
    int radius;

    MoveTo(hDC, np->clockCenter.x, np->clockCenter.y);
    radius = MulDiv(np->clockRadius, scale, 100);
    lppt = rCircleTable + pos;
    SetROP2(hDC, patMode);
    SelectObject(hDC, hPen);

    LineTo( hDC,
            np->clockCenter.x + MulDiv(lppt->x, radius, 8000),
            np->clockCenter.y + MulDiv(lppt->y, radius, 8000) );
}

```

Теперь всё ясно: координаты были предвычислены, как если бы циферблат был размером  $2 \cdot 8000$ , а затем он масштабируется до радиуса текущего циферблата используя ф-цию *MulDiv()*.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Структура POINT<sup>10</sup> это структура из двух 32-битных значений, первое это x, второе это y.

## 8.6. Донглы

Автор этих строк иногда делал замену **донглам** или «эмуляторы донглов» и здесь немного примеров, как это происходит.

Об одном неопisanном здесь случае с Rockey и Z3 вы также можете прочитать здесь : [http://yurichev.com/tmp/SAT\\_SMT\\_DRAFT.pdf](http://yurichev.com/tmp/SAT_SMT_DRAFT.pdf).

### 8.6.1. Пример #1: MacOS Classic и PowerPC

Вот пример программы для MacOS Classic<sup>11</sup>, для PowerPC. Компания, разработавшая этот продукт, давно исчезла, так что (легальный) пользователь боялся того что донгла может сломаться.

Если запустить программу без подключенной донглы, можно увидеть окно с надписью

“Invalid Security Device”. Мне повезло потому что этот текст можно было легко найти внутри исполняемого файла.

Представим, что мы не знакомы ни с Mac OS Classic, ни с PowerPC, но всё-таки попробуем.

**IDA!** открывает исполняемый файл легко, показывая его тип как

“PEF (Mac OS or Be OS executable)” (действительно, это стандартный тип файлов в Mac OS Classic).

В поисках текстовой строки с сообщением об ошибке, мы попадаем на этот фрагмент кода:

```
...
seg000:000C87FC 38 60 00 01  li    %r3, 1
seg000:000C8800 48 03 93 41  bl    check1
seg000:000C8804 60 00 00 00  nop
seg000:000C8808 54 60 06 3F  clrlwi. %r0, %r3, 24
seg000:000C880C 40 82 00 40  bne   OK
seg000:000C8810 80 62 9F D8  lwz   %r3, TC_aInvalidSecurityDevice
...
```

Да, это код PowerPC. Это очень типичный процессор для **RISC!** 1990-х. Каждая инструкция занимает 4 байта (как и в MIPS и ARM) и их имена немного похожи на имена инструкций MIPS.

<sup>10</sup>[https://msdn.microsoft.com/en-us/library/windows/desktop/dd162805\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dd162805(v=vs.85).aspx)

<sup>11</sup>MacOS перед тем как перейти на UNIX

check1() это имя которое мы дадим этой функции немного позже. BL это инструкция *Branch Link* т.е. предназначенная для вызова подпрограмм. Самое важное место — это инструкция **BNE!**, срабатывающая, если проверка наличия донглы прошла успешно, либо не срабатывающая в случае ошибки: и тогда адрес текстовой строки с сообщением об ошибке будет загружен в регистр r3 для последующей передачи в функцию отображения диалогового окна.

Из [Steve Zucker, SunSoft and Kari Karhi, IBM, *SYSTEM V APPLICATION BINARY INTERFACE: PowerPC Processor Supplement*, (1995)]<sup>12</sup> мы узнаем, что регистр r3 используется для возврата значений (и еще r4 если значение 64-битное).

Еще одна, пока что неизвестная инструкция CLRLWI. Из [*PowerPC(tm) Microprocessor Family: The Programming Environments for 32-Bit Microprocessors*, (2000)]<sup>13</sup> мы узнаем, что эта инструкция одновременно и очищает и загружает. В нашем случае, она очищает 24 старших бита из значения в r3 и записывает всё это в r0, так что это аналог MOVZX в x86 (?? (стр. ??)), но также устанавливает флаги, так что **BNE!** может проверить их потом.

Посмотрим внутрь check1():

```

seg000:00101B40      check1: # CODE XREF: seg000:00063E7Cp
seg000:00101B40      # sub_64070+160p ...
seg000:00101B40
seg000:00101B40      .set arg_8, 8
seg000:00101B40
seg000:00101B40 7C 08 02 A6      mflr    %r0
seg000:00101B44 90 01 00 08      stw     %r0, arg_8(%sp)
seg000:00101B48 94 21 FF C0      stwu   %sp, -0x40(%sp)
seg000:00101B4C 48 01 6B 39      bl     check2
seg000:00101B50 60 00 00 00      nop
seg000:00101B54 80 01 00 48      lwz    %r0, 0x40+arg_8(%sp)
seg000:00101B58 38 21 00 40      addi   %sp, %sp, 0x40
seg000:00101B5C 7C 08 03 A6      mtlr   %r0
seg000:00101B60 4E 80 00 20      blr
seg000:00101B60      # End of function check1

```

Как можно увидеть в **IDA!**, эта функция вызывается из многих мест в программе, но только значение в регистре r3 проверяется сразу после каждого вызова. Всё что эта функция делает это только вызывает другую функцию, так что это **thunk function**: здесь присутствует и пролог функции и эпилог, но регистр r3 не трогается, так что check1() возвращает то, что возвращает check2().

**BLR!**<sup>14</sup> это похоже возврат из функции, но так как IDA делает всю разметку функций автоматически, наверное, мы можем пока не интересоваться этим. Так как это типичный **RISC!**, похоже, подпрограммы вызываются, используя **link register**, точно как в ARM.

Функция check2() более сложная:

```

seg000:00118684      check2: # CODE XREF: check1+Cp

```

<sup>12</sup>Также доступно здесь: [http://yurichev.com/mirrors/PowerPC/elfspec\\_ppc.pdf](http://yurichev.com/mirrors/PowerPC/elfspec_ppc.pdf)

<sup>13</sup>Также доступно здесь: [http://yurichev.com/mirrors/PowerPC/6xx\\_pem.pdf](http://yurichev.com/mirrors/PowerPC/6xx_pem.pdf)

<sup>14</sup>**BLR!**

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

seg000:00118684
seg000:00118684      .set var_18, -0x18
seg000:00118684      .set var_C, -0xC
seg000:00118684      .set var_8, -8
seg000:00118684      .set var_4, -4
seg000:00118684      .set arg_8, 8
seg000:00118684 93 E1 FF FC stw    %r31, var_4(%sp)
seg000:00118688 7C 08 02 A6 mflr   %r0
seg000:0011868C 83 E2 95 A8 lwz    %r31, off_1485E8 # dword_24B704
seg000:00118690      .using dword_24B704, %r31
seg000:00118690 93 C1 FF F8 stw    %r30, var_8(%sp)
seg000:00118694 93 A1 FF F4 stw    %r29, var_C(%sp)
seg000:00118698 7C 7D 1B 78 mr     %r29, %r3
seg000:0011869C 90 01 00 08 stw    %r0, arg_8(%sp)
seg000:001186A0 54 60 06 3E clrlwi %r0, %r3, 24
seg000:001186A4 28 00 00 01 cmplwi %r0, 1
seg000:001186A8 94 21 FF B0 stwu   %sp, -0x50(%sp)
seg000:001186AC 40 82 00 0C bne    loc_1186B8
seg000:001186B0 38 60 00 01 li     %r3, 1
seg000:001186B4 48 00 00 6C b      exit
seg000:001186B8
seg000:001186B8      loc_1186B8: # CODE XREF: check2+28j
seg000:001186B8 48 00 03 D5 bl     sub_118A8C
seg000:001186BC 60 00 00 00 nop
seg000:001186C0 3B C0 00 00 li     %r30, 0
seg000:001186C4
seg000:001186C4      skip:      # CODE XREF: check2+94j
seg000:001186C4 57 C0 06 3F clrlwi. %r0, %r30, 24
seg000:001186C8 41 82 00 18 beq    loc_1186E0
seg000:001186CC 38 61 00 38 addi   %r3, %sp, 0x50+var_18
seg000:001186D0 80 9F 00 00 lwz    %r4, dword_24B704
seg000:001186D4 48 00 C0 55 bl     .RBEFINDNEXT
seg000:001186D8 60 00 00 00 nop
seg000:001186DC 48 00 00 1C b      loc_1186F8
seg000:001186E0
seg000:001186E0      loc_1186E0: # CODE XREF: check2+44j
seg000:001186E0 80 BF 00 00 lwz    %r5, dword_24B704
seg000:001186E4 38 81 00 38 addi   %r4, %sp, 0x50+var_18
seg000:001186E8 38 60 08 C2 li     %r3, 0x1234
seg000:001186EC 48 00 BF 99 bl     .RBEFINDFIRST
seg000:001186F0 60 00 00 00 nop
seg000:001186F4 3B C0 00 01 li     %r30, 1
seg000:001186F8
seg000:001186F8      loc_1186F8: # CODE XREF: check2+58j
seg000:001186F8 54 60 04 3F clrlwi. %r0, %r3, 16
seg000:001186FC 41 82 00 0C beq    must_jump
seg000:00118700 38 60 00 00 li     %r3, 0 # error
seg000:00118704 48 00 00 1C b      exit
seg000:00118708
seg000:00118708      must_jump: # CODE XREF: check2+78j
seg000:00118708 7F A3 EB 78 mr     %r3, %r29
seg000:0011870C 48 00 00 31 bl     check3

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

seg000:00118710 60 00 00 00  nop
seg000:00118714 54 60 06 3F  clrlwi. %r0, %r3, 24
seg000:00118718 41 82 FF AC  beq     skip
seg000:0011871C 38 60 00 01  li     %r3, 1
seg000:00118720
seg000:00118720          exit:      # CODE XREF: check2+30j
seg000:00118720          # check2+80j
seg000:00118720 80 01 00 58  lwz   %r0, 0x50+arg_8(%sp)
seg000:00118724 38 21 00 50  addi  %sp, %sp, 0x50
seg000:00118728 83 E1 FF FC  lwz   %r31, var_4(%sp)
seg000:0011872C 7C 08 03 A6  mtlr  %r0
seg000:00118730 83 C1 FF F8  lwz   %r30, var_8(%sp)
seg000:00118734 83 A1 FF F4  lwz   %r29, var_C(%sp)
seg000:00118738 4E 80 00 20  blr
seg000:00118738          # End of function check2

```

Снова повезло: имена некоторых функций оставлены в исполняемом файле (в символах в отладочной секции? Трудно сказать до тех пор, пока мы не знакомы с этим форматом файлов, может быть это что-то вроде PE-экспортов (??))? Как например `.RBEFINDNEXT()` and `.RBEFINDFIRST()`.

В итоге, эти функции вызывают другие функции с именами вроде `.GetNextDeviceViaUSB()`, `.USBSendPKT()`, так что они явно работают с каким-то USB-устройством.

Тут даже есть функция с названием `.GetNextEve3Device()` — звучит знакомо, в 1990-х годах была донгла Sentinel Eve3 для ADB-порта (присутствующих на Макинтошах).

В начале посмотрим на то как устанавливается регистр `r3` одновременно игнорируя всё остальное. Мы знаем, что «хорошее» значение в `r3` должно быть не нулевым, а нулевой `r3` приведет к выводу диалогового окна с сообщением об ошибке.

В функции имеются две инструкции `li %r3, 1` и одна `li %r3, 0` (*Load Immediate*, т.е. загрузить значение в регистр). Самая первая инструкция находится на `0x001186B0` — и честно говоря, трудно заранее понять, что это означает.

А вот то что мы видим дальше понять проще: вызывается `.RBEFINDFIRST()` и в случае ошибки, `0` будет записан в `r3` и мы перейдем на `exit`, а иначе будет вызвана функция `check3()` — если и она будет выполнена с ошибкой, будет вызвана `.RBEFINDNEXT()` вероятно, для поиска другого USB-устройства.

N.B.: `clrlwi. %r0, %r3, 16` это аналог того что мы уже видели, но она очищает 16 старших бит, т.е., `.RBEFINDFIRST()` вероятно возвращает 16-битное значение.

`B` (означает *branch*) — безусловный переход.

**BEQ!** это обратная инструкция от **BNE!**.

Посмотрим на `check3()`:

```

seg000:0011873C          check3: # CODE XREF: check2+88p
seg000:0011873C
seg000:0011873C          .set var_18, -0x18

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

seg000:0011873C      .set var_C, -0xC
seg000:0011873C      .set var_8, -8
seg000:0011873C      .set var_4, -4
seg000:0011873C      .set arg_8, 8
seg000:0011873C
seg000:0011873C 93 E1 FF FC      stw    %r31, var_4(%sp)
seg000:00118740 7C 08 02 A6      mflr  %r0
seg000:00118744 38 A0 00 00      li    %r5, 0
seg000:00118748 93 C1 FF F8      stw    %r30, var_8(%sp)
seg000:0011874C 83 C2 95 A8      lwz   %r30, off_1485E8 # dword_24B704
seg000:00118750      .using dword_24B704, %r30
seg000:00118750 93 A1 FF F4      stw    %r29, var_C(%sp)
seg000:00118754 3B A3 00 00      addi  %r29, %r3, 0
seg000:00118758 38 60 00 00      li    %r3, 0
seg000:0011875C 90 01 00 08      stw    %r0, arg_8(%sp)
seg000:00118760 94 21 FF B0      stwu  %sp, -0x50(%sp)
seg000:00118764 80 DE 00 00      lwz   %r6, dword_24B704
seg000:00118768 38 81 00 38      addi  %r4, %sp, 0x50+var_18
seg000:0011876C 48 00 C0 5D      bl    .RBEREAD
seg000:00118770 60 00 00 00      nop
seg000:00118774 54 60 04 3F      clrlwi. %r0, %r3, 16
seg000:00118778 41 82 00 0C      beq   loc_118784
seg000:0011877C 38 60 00 00      li    %r3, 0
seg000:00118780 48 00 02 F0      b     exit
seg000:00118784
seg000:00118784      loc_118784: # CODE XREF: check3+3Cj
seg000:00118784 A0 01 00 38      lhz   %r0, 0x50+var_18(%sp)
seg000:00118788 28 00 04 B2      cmplwi %r0, 0x1100
seg000:0011878C 41 82 00 0C      beq   loc_118798
seg000:00118790 38 60 00 00      li    %r3, 0
seg000:00118794 48 00 02 DC      b     exit
seg000:00118798
seg000:00118798      loc_118798: # CODE XREF: check3+50j
seg000:00118798 80 DE 00 00      lwz   %r6, dword_24B704
seg000:0011879C 38 81 00 38      addi  %r4, %sp, 0x50+var_18
seg000:001187A0 38 60 00 01      li    %r3, 1
seg000:001187A4 38 A0 00 00      li    %r5, 0
seg000:001187A8 48 00 C0 21      bl    .RBEREAD
seg000:001187AC 60 00 00 00      nop
seg000:001187B0 54 60 04 3F      clrlwi. %r0, %r3, 16
seg000:001187B4 41 82 00 0C      beq   loc_1187C0
seg000:001187B8 38 60 00 00      li    %r3, 0
seg000:001187BC 48 00 02 B4      b     exit
seg000:001187C0
seg000:001187C0      loc_1187C0: # CODE XREF: check3+78j
seg000:001187C0 A0 01 00 38      lhz   %r0, 0x50+var_18(%sp)
seg000:001187C4 28 00 06 4B      cmplwi %r0, 0x09AB
seg000:001187C8 41 82 00 0C      beq   loc_1187D4
seg000:001187CC 38 60 00 00      li    %r3, 0
seg000:001187D0 48 00 02 A0      b     exit
seg000:001187D4
seg000:001187D4      loc_1187D4: # CODE XREF: check3+8Cj
seg000:001187D4 4B F9 F3 D9      bl    sub_B7BAC

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

seg000:001187D8 60 00 00 00 nop
seg000:001187DC 54 60 06 3E clrlwi %r0, %r3, 24
seg000:001187E0 2C 00 00 05 cmpwi %r0, 5
seg000:001187E4 41 82 01 00 beq loc_1188E4
seg000:001187E8 40 80 00 10 bge loc_1187F8
seg000:001187EC 2C 00 00 04 cmpwi %r0, 4
seg000:001187F0 40 80 00 58 bge loc_118848
seg000:001187F4 48 00 01 8C b loc_118980
seg000:001187F8
seg000:001187F8 loc_1187F8: # CODE XREF: check3+ACj
seg000:001187F8 2C 00 00 0B cmpwi %r0, 0xB
seg000:001187FC 41 82 00 08 beq loc_118804
seg000:00118800 48 00 01 80 b loc_118980
seg000:00118804
seg000:00118804 loc_118804: # CODE XREF: check3+C0j
seg000:00118804 80 DE 00 00 lwz %r6, dword_24B704
seg000:00118808 38 81 00 38 addi %r4, %sp, 0x50+var_18
seg000:0011880C 38 60 00 08 li %r3, 8
seg000:00118810 38 A0 00 00 li %r5, 0
seg000:00118814 48 00 BF B5 bl .RBEREAD
seg000:00118818 60 00 00 00 nop
seg000:0011881C 54 60 04 3F clrlwi. %r0, %r3, 16
seg000:00118820 41 82 00 0C beq loc_11882C
seg000:00118824 38 60 00 00 li %r3, 0
seg000:00118828 48 00 02 48 b exit
seg000:0011882C
seg000:0011882C loc_11882C: # CODE XREF: check3+E4j
seg000:0011882C A0 01 00 38 lhz %r0, 0x50+var_18(%sp)
seg000:00118830 28 00 11 30 cmplwi %r0, 0xFEAO
seg000:00118834 41 82 00 0C beq loc_118840
seg000:00118838 38 60 00 00 li %r3, 0
seg000:0011883C 48 00 02 34 b exit
seg000:00118840
seg000:00118840 loc_118840: # CODE XREF: check3+F8j
seg000:00118840 38 60 00 01 li %r3, 1
seg000:00118844 48 00 02 2C b exit
seg000:00118848
seg000:00118848 loc_118848: # CODE XREF: check3+B4j
seg000:00118848 80 DE 00 00 lwz %r6, dword_24B704
seg000:0011884C 38 81 00 38 addi %r4, %sp, 0x50+var_18
seg000:00118850 38 60 00 0A li %r3, 0xA
seg000:00118854 38 A0 00 00 li %r5, 0
seg000:00118858 48 00 BF 71 bl .RBEREAD
seg000:0011885C 60 00 00 00 nop
seg000:00118860 54 60 04 3F clrlwi. %r0, %r3, 16
seg000:00118864 41 82 00 0C beq loc_118870
seg000:00118868 38 60 00 00 li %r3, 0
seg000:0011886C 48 00 02 04 b exit
seg000:00118870
seg000:00118870 loc_118870: # CODE XREF: check3+128j
seg000:00118870 A0 01 00 38 lhz %r0, 0x50+var_18(%sp)
seg000:00118874 28 00 03 F3 cmplwi %r0, 0xA6E1
seg000:00118878 41 82 00 0C beq loc_118884

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мой адреса](#). Спасибо!

```

seg000:0011887C 38 60 00 00  li    %r3, 0
seg000:00118880 48 00 01 F0  b     exit
seg000:00118884
seg000:00118884          loc_118884: # CODE XREF: check3+13Cj
seg000:00118884 57 BF 06 3E  clrlwi %r31, %r29, 24
seg000:00118888 28 1F 00 02  cmlpli %r31, 2
seg000:0011888C 40 82 00 0C  bne   loc_118898
seg000:00118890 38 60 00 01  li    %r3, 1
seg000:00118894 48 00 01 DC  b     exit
seg000:00118898
seg000:00118898          loc_118898: # CODE XREF: check3+150j
seg000:00118898 80 DE 00 00  lwz   %r6, dword_24B704
seg000:0011889C 38 81 00 38  addi  %r4, %sp, 0x50+var_18
seg000:001188A0 38 60 00 0B  li    %r3, 0xB
seg000:001188A4 38 A0 00 00  li    %r5, 0
seg000:001188A8 48 00 BF 21  bl    .RBEREAD
seg000:001188AC 60 00 00 00  nop
seg000:001188B0 54 60 04 3F  clrlwi. %r0, %r3, 16
seg000:001188B4 41 82 00 0C  beq   loc_1188C0
seg000:001188B8 38 60 00 00  li    %r3, 0
seg000:001188BC 48 00 01 B4  b     exit
seg000:001188C0
seg000:001188C0          loc_1188C0: # CODE XREF: check3+178j
seg000:001188C0 A0 01 00 38  lhz   %r0, 0x50+var_18(%sp)
seg000:001188C4 28 00 23 1C  cmlpli %r0, 0x1C20
seg000:001188C8 41 82 00 0C  beq   loc_1188D4
seg000:001188CC 38 60 00 00  li    %r3, 0
seg000:001188D0 48 00 01 A0  b     exit
seg000:001188D4
seg000:001188D4          loc_1188D4: # CODE XREF: check3+18Cj
seg000:001188D4 28 1F 00 03  cmlpli %r31, 3
seg000:001188D8 40 82 01 94  bne   error
seg000:001188DC 38 60 00 01  li    %r3, 1
seg000:001188E0 48 00 01 90  b     exit
seg000:001188E4
seg000:001188E4          loc_1188E4: # CODE XREF: check3+A8j
seg000:001188E4 80 DE 00 00  lwz   %r6, dword_24B704
seg000:001188E8 38 81 00 38  addi  %r4, %sp, 0x50+var_18
seg000:001188EC 38 60 00 0C  li    %r3, 0xC
seg000:001188F0 38 A0 00 00  li    %r5, 0
seg000:001188F4 48 00 BE D5  bl    .RBEREAD
seg000:001188F8 60 00 00 00  nop
seg000:001188FC 54 60 04 3F  clrlwi. %r0, %r3, 16
seg000:00118900 41 82 00 0C  beq   loc_11890C
seg000:00118904 38 60 00 00  li    %r3, 0
seg000:00118908 48 00 01 68  b     exit
seg000:0011890C
seg000:0011890C          loc_11890C: # CODE XREF: check3+1C4j
seg000:0011890C A0 01 00 38  lhz   %r0, 0x50+var_18(%sp)
seg000:00118910 28 00 1F 40  cmlpli %r0, 0x40FF
seg000:00118914 41 82 00 0C  beq   loc_118920
seg000:00118918 38 60 00 00  li    %r3, 0
seg000:0011891C 48 00 01 54  b     exit

```

```

seg000:00118920
seg000:00118920      loc_118920: # CODE XREF: check3+1D8j
seg000:00118920 57 BF 06 3E      clrlwi  %r31, %r29, 24
seg000:00118924 28 1F 00 02      cmplwi  %r31, 2
seg000:00118928 40 82 00 0C      bne     loc_118934
seg000:0011892C 38 60 00 01      li     %r3, 1
seg000:00118930 48 00 01 40      b      exit
seg000:00118934
seg000:00118934      loc_118934: # CODE XREF: check3+1ECj
seg000:00118934 80 DE 00 00      lwz    %r6, dword_24B704
seg000:00118938 38 81 00 38      addi   %r4, %sp, 0x50+var_18
seg000:0011893C 38 60 00 0D      li     %r3, 0xD
seg000:00118940 38 A0 00 00      li     %r5, 0
seg000:00118944 48 00 BE 85      bl     .RBEREAD
seg000:00118948 60 00 00 00      nop
seg000:0011894C 54 60 04 3F      clrlwi. %r0, %r3, 16
seg000:00118950 41 82 00 0C      beq    loc_11895C
seg000:00118954 38 60 00 00      li     %r3, 0
seg000:00118958 48 00 01 18      b      exit
seg000:0011895C
seg000:0011895C      loc_11895C: # CODE XREF: check3+214j
seg000:0011895C A0 01 00 38      lhz    %r0, 0x50+var_18(%sp)
seg000:00118960 28 00 07 CF      cmplwi %r0, 0xFC7
seg000:00118964 41 82 00 0C      beq    loc_118970
seg000:00118968 38 60 00 00      li     %r3, 0
seg000:0011896C 48 00 01 04      b      exit
seg000:00118970
seg000:00118970      loc_118970: # CODE XREF: check3+228j
seg000:00118970 28 1F 00 03      cmplwi %r31, 3
seg000:00118974 40 82 00 F8      bne    error
seg000:00118978 38 60 00 01      li     %r3, 1
seg000:0011897C 48 00 00 F4      b      exit
seg000:00118980
seg000:00118980      loc_118980: # CODE XREF: check3+B8j
seg000:00118980      # check3+C4j
seg000:00118980 80 DE 00 00      lwz    %r6, dword_24B704
seg000:00118984 38 81 00 38      addi   %r4, %sp, 0x50+var_18
seg000:00118988 3B E0 00 00      li     %r31, 0
seg000:0011898C 38 60 00 04      li     %r3, 4
seg000:00118990 38 A0 00 00      li     %r5, 0
seg000:00118994 48 00 BE 35      bl     .RBEREAD
seg000:00118998 60 00 00 00      nop
seg000:0011899C 54 60 04 3F      clrlwi. %r0, %r3, 16
seg000:001189A0 41 82 00 0C      beq    loc_1189AC
seg000:001189A4 38 60 00 00      li     %r3, 0
seg000:001189A8 48 00 00 C8      b      exit
seg000:001189AC
seg000:001189AC      loc_1189AC: # CODE XREF: check3+264j
seg000:001189AC A0 01 00 38      lhz    %r0, 0x50+var_18(%sp)
seg000:001189B0 28 00 1D 6A      cmplwi %r0, 0xAED0
seg000:001189B4 40 82 00 0C      bne    loc_1189C0
seg000:001189B8 3B E0 00 01      li     %r31, 1
seg000:001189BC 48 00 00 14      b      loc_1189D0

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

seg000:001189C0
seg000:001189C0          loc_1189C0: # CODE XREF: check3+278j
seg000:001189C0 28 00 18 28    cmplwi  %r0, 0x2818
seg000:001189C4 41 82 00 0C    beq     loc_1189D0
seg000:001189C8 38 60 00 00    li     %r3, 0
seg000:001189CC 48 00 00 A4    b      exit
seg000:001189D0
seg000:001189D0          loc_1189D0: # CODE XREF: check3+280j
seg000:001189D0          # check3+288j
seg000:001189D0 57 A0 06 3E    clrlwi %r0, %r29, 24
seg000:001189D4 28 00 00 02    cmplwi %r0, 2
seg000:001189D8 40 82 00 20    bne    loc_1189F8
seg000:001189DC 57 E0 06 3F    clrlwi %r0, %r31, 24
seg000:001189E0 41 82 00 10    beq    good2
seg000:001189E4 48 00 4C 69    bl     sub_11D64C
seg000:001189E8 60 00 00 00    nop
seg000:001189EC 48 00 00 84    b      exit
seg000:001189F0
seg000:001189F0          good2:      # CODE XREF: check3+2A4j
seg000:001189F0 38 60 00 01    li     %r3, 1
seg000:001189F4 48 00 00 7C    b      exit
seg000:001189F8
seg000:001189F8          loc_1189F8: # CODE XREF: check3+29Cj
seg000:001189F8 80 DE 00 00    lwz   %r6, dword_24B704
seg000:001189FC 38 81 00 38    addi  %r4, %sp, 0x50+var_18
seg000:00118A00 38 60 00 05    li     %r3, 5
seg000:00118A04 38 A0 00 00    li     %r5, 0
seg000:00118A08 48 00 BD C1    bl     .RBEREAD
seg000:00118A0C 60 00 00 00    nop
seg000:00118A10 54 60 04 3F    clrlwi %r0, %r3, 16
seg000:00118A14 41 82 00 0C    beq    loc_118A20
seg000:00118A18 38 60 00 00    li     %r3, 0
seg000:00118A1C 48 00 00 54    b      exit
seg000:00118A20
seg000:00118A20          loc_118A20: # CODE XREF: check3+2D8j
seg000:00118A20 A0 01 00 38    lhz   %r0, 0x50+var_18(%sp)
seg000:00118A24 28 00 11 D3    cmplwi %r0, 0xD300
seg000:00118A28 40 82 00 0C    bne    loc_118A34
seg000:00118A2C 3B E0 00 01    li     %r31, 1
seg000:00118A30 48 00 00 14    b      good1
seg000:00118A34
seg000:00118A34          loc_118A34: # CODE XREF: check3+2ECj
seg000:00118A34 28 00 1A EB    cmplwi %r0, 0xEBA1
seg000:00118A38 41 82 00 0C    beq    good1
seg000:00118A3C 38 60 00 00    li     %r3, 0
seg000:00118A40 48 00 00 30    b      exit
seg000:00118A44
seg000:00118A44          good1:      # CODE XREF: check3+2F4j
seg000:00118A44          # check3+2FCj
seg000:00118A44 57 A0 06 3E    clrlwi %r0, %r29, 24
seg000:00118A48 28 00 00 03    cmplwi %r0, 3
seg000:00118A4C 40 82 00 20    bne    error
seg000:00118A50 57 E0 06 3F    clrlwi %r0, %r31, 24

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

seg000:00118A54 41 82 00 10    beq    good
seg000:00118A58 48 00 4B F5    bl     sub_11D64C
seg000:00118A5C 60 00 00 00    nop
seg000:00118A60 48 00 00 10    b      exit
seg000:00118A64
seg000:00118A64                good:    # CODE XREF: check3+318j
seg000:00118A64 38 60 00 01    li     %r3, 1
seg000:00118A68 48 00 00 08    b      exit
seg000:00118A6C
seg000:00118A6C                error:  # CODE XREF: check3+19Cj
seg000:00118A6C                # check3+238j ...
seg000:00118A6C 38 60 00 00    li     %r3, 0
seg000:00118A70
seg000:00118A70                exit:   # CODE XREF: check3+44j
seg000:00118A70                # check3+58j ...
seg000:00118A70 80 01 00 58    lwz   %r0, 0x50+arg_8(%sp)
seg000:00118A74 38 21 00 50    addi  %sp, %sp, 0x50
seg000:00118A78 83 E1 FF FC    lwz   %r31, var_4(%sp)
seg000:00118A7C 7C 08 03 A6    mtlr  %r0
seg000:00118A80 83 C1 FF F8    lwz   %r30, var_8(%sp)
seg000:00118A84 83 A1 FF F4    lwz   %r29, var_C(%sp)
seg000:00118A88 4E 80 00 20    blr
seg000:00118A88                # End of function check3

```

Здесь много вызовов `.RBEREAD()`. Эта функция, должно быть, читает какие-то значения из донглы, которые потом сравниваются здесь при помощи `CMPLWI`.

Мы также видим в регистр `r3` записывается перед каждым вызовом `.RBEREAD()` одно из этих значений: 0, 1, 8, 0xA, 0xB, 0xC, 0xD, 4, 5. Вероятно адрес в памяти или что-то в этом роде?

Да, действительно, если погуглить имена этих функций, можно легко найти документацию к Sentinel Eve3!

Наверное, уже и не нужно изучать остальные инструкции PowerPC: всё что делает эта функция это просто вызывает `.RBEREAD()`, сравнивает его результаты с константами и возвращает 1 если результат сравнения положительный или 0 в другом случае.

Всё ясно: `check1()` должна всегда возвращать 1 или иное ненулевое значение. Но так как мы не очень уверены в своих знаниях инструкций PowerPC, будем осторожны и пропатчим переходы в `check2` на адресах `0x001186FC` и `0x00118718`.

На `0x001186FC` мы записываем байты `0x48` и `0` таким образом превращая инструкцию **BEQ!** в инструкцию **B** (безусловный переход): мы можем заметить этот опкод прямо в коде даже без обращения к *[PowerPC(tm) Microprocessor Family: The Programming Environments for 32-Bit Microprocessors, (2000)]*<sup>15</sup>.

На `0x00118718` мы записываем байт `0x60` и еще 3 нулевых байта, таким образом превращая её в инструкцию **NOP!**: Этот опкод мы также можем подсмотреть прямо в коде.

<sup>15</sup>Также доступно здесь: [http://yurichev.com/mirrors/PowerPC/6xx\\_pem.pdf](http://yurichev.com/mirrors/PowerPC/6xx_pem.pdf)

И всё заработало без подключенной донглы.

Резюмируя, такие простые модификации можно делать в **IDA!** даже с минимальными знаниями ассемблера.

## 8.6.2. Пример #2: SCO OpenServer

Древняя программа для SCO OpenServer от 1997 разработанная давно исчезнувшей компанией.

Специальный драйвер донглы инсталлируется в системе, он содержит такие текстовые строки: «Copyright 1989, Rainbow Technologies, Inc., Irvine, CA» и «Sentinel Integrated Driver Ver. 3.0».

После инсталляции драйвера, в /dev появляются такие устройства:

```
/dev/rbsl8
/dev/rbsl9
/dev/rbsl10
```

Без подключенной донглы, программа сообщает об ошибке, но сообщение об ошибке не удается найти в исполняемых файлах.

Еще раз спасибо **IDA!**, она легко загружает исполняемые файлы формата COFF использующиеся в SCO OpenServer.

Попробуем также поискать строку «rbsl», и действительно, её можно найти в таком фрагменте кода:

```
.text:00022AB8      public SSQC
.text:00022AB8 SSQC  proc near ; CODE XREF: SSQ+7p
.text:00022AB8
.text:00022AB8 var_44 = byte ptr -44h
.text:00022AB8 var_29 = byte ptr -29h
.text:00022AB8 arg_0  = dword ptr 8
.text:00022AB8
.text:00022AB8      push     ebp
.text:00022AB9      mov      ebp, esp
.text:00022ABB      sub      esp, 44h
.text:00022ABE      push     edi
.text:00022ABF      mov      edi, offset unk_4035D0
.text:00022AC4      push     esi
.text:00022AC5      mov      esi, [ebp+arg_0]
.text:00022AC8      push     ebx
.text:00022AC9      push     esi
.text:00022ACA      call    strlen
.text:00022ACF      add      esp, 4
.text:00022AD2      cmp      eax, 2
.text:00022AD7      jnz     loc_22BA4
.text:00022ADD      inc      esi
.text:00022ADE      mov      al, [esi-1]
.text:00022AE1      movsx   eax, al
.text:00022AE4      cmp      eax, '3'
.text:00022AE9      jz      loc_22B84
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:00022AEF      cmp     eax, '4'
.text:00022AF4      jz      loc_22B94
.text:00022AFA      cmp     eax, '5'
.text:00022AFF      jnz     short loc_22B6B
.text:00022B01      movsx   ebx, byte ptr [esi]
.text:00022B04      sub     ebx, '0'
.text:00022B07      mov     eax, 7
.text:00022B0C      add     eax, ebx
.text:00022B0E      push   eax
.text:00022B0F      lea    eax, [ebp+var_44]
.text:00022B12      push   offset aDevSlD ; "/dev/sl%d"
.text:00022B17      push   eax
.text:00022B18      call   nl_sprintf
.text:00022B1D      push   0 ; int
.text:00022B1F      push   offset aDevRbsl8 ; char *
.text:00022B24      call   _access
.text:00022B29      add     esp, 14h
.text:00022B2C      cmp     eax, 0FFFFFFFh
.text:00022B31      jz      short loc_22B48
.text:00022B33      lea    eax, [ebx+7]
.text:00022B36      push   eax
.text:00022B37      lea    eax, [ebp+var_44]
.text:00022B3A      push   offset aDevRbslD ; "/dev/rbsl%d"
.text:00022B3F      push   eax
.text:00022B40      call   nl_sprintf
.text:00022B45      add     esp, 0Ch
.text:00022B48
.text:00022B48  loc_22B48: ; CODE XREF: SSQC+79j
.text:00022B48      mov     edx, [edi]
.text:00022B4A      test    edx, edx
.text:00022B4C      jle    short loc_22B57
.text:00022B4E      push   edx ; int
.text:00022B4F      call   _close
.text:00022B54      add     esp, 4
.text:00022B57
.text:00022B57  loc_22B57: ; CODE XREF: SSQC+94j
.text:00022B57      push   2 ; int
.text:00022B59      lea    eax, [ebp+var_44]
.text:00022B5C      push   eax ; char *
.text:00022B5D      call   _open
.text:00022B62      add     esp, 8
.text:00022B65      test    eax, eax
.text:00022B67      mov     [edi], eax
.text:00022B69      jge    short loc_22B78
.text:00022B6B
.text:00022B6B  loc_22B6B: ; CODE XREF: SSQC+47j
.text:00022B6B      mov     eax, 0FFFFFFFh
.text:00022B70      pop     ebx
.text:00022B71      pop     esi
.text:00022B72      pop     edi
.text:00022B73      mov     esp, ebp
.text:00022B75      pop     ebp
.text:00022B76      retn

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

.text:00022B78
.text:00022B78 loc_22B78: ; CODE XREF: SSQC+B1j
                pop     ebx
.text:00022B79                pop     esi
.text:00022B7A                pop     edi
.text:00022B7B                xor     eax, eax
.text:00022B7D                mov     esp, ebp
.text:00022B7F                pop     ebp
.text:00022B80                retn
.text:00022B84
.text:00022B84 loc_22B84: ; CODE XREF: SSQC+31j
                mov     al, [esi]
.text:00022B86                pop     ebx
.text:00022B87                pop     esi
.text:00022B88                pop     edi
.text:00022B89                mov     ds:byte_407224, al
.text:00022B8E                mov     esp, ebp
.text:00022B90                xor     eax, eax
.text:00022B92                pop     ebp
.text:00022B93                retn
.text:00022B94
.text:00022B94 loc_22B94: ; CODE XREF: SSQC+3Cj
                mov     al, [esi]
.text:00022B96                pop     ebx
.text:00022B97                pop     esi
.text:00022B98                pop     edi
.text:00022B99                mov     ds:byte_407225, al
.text:00022B9E                mov     esp, ebp
.text:00022BA0                xor     eax, eax
.text:00022BA2                pop     ebp
.text:00022BA3                retn
.text:00022BA4
.text:00022BA4 loc_22BA4: ; CODE XREF: SSQC+1Fj
                movsx  eax, ds:byte_407225
.text:00022BAB                push   esi
.text:00022BAC                push   eax
.text:00022BAD                movsx  eax, ds:byte_407224
.text:00022BB4                push   eax
.text:00022BB5                lea   eax, [ebp+var_44]
.text:00022BB8                push   offset a46CC5 ; "46%c%c%s"
.text:00022BBD                push   eax
.text:00022BBE                call  nl_sprintf
.text:00022BC3                lea   eax, [ebp+var_44]
.text:00022BC6                push   eax
.text:00022BC7                call  strlen
.text:00022BCC                add   esp, 18h
.text:00022BCF                cmp   eax, 1Bh
.text:00022BD4                jle   short loc_22BDA
.text:00022BD6                mov   [ebp+var_29], 0
.text:00022BDA
.text:00022BDA loc_22BDA: ; CODE XREF: SSQC+11Cj
                lea   eax, [ebp+var_44]
.text:00022BDD                push   eax

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

.text:00022BDE      call     strlen
.text:00022BE3      push    eax                ; unsigned int
.text:00022BE4      lea    eax, [ebp+var_44]
.text:00022BE7      push    eax                ; void *
.text:00022BE8      mov    eax, [edi]
.text:00022BEA      push    eax                ; int
.text:00022BEB      call   _write
.text:00022BF0      add    esp, 10h
.text:00022BF3      pop    ebx
.text:00022BF4      pop    esi
.text:00022BF5      pop    edi
.text:00022BF6      mov    esp, ebp
.text:00022BF8      pop    ebp
.text:00022BF9      retn
.text:00022BFA      db 0Eh dup(90h)
.text:00022BFA SSQC endp

```

Действительно, должна же как-то программа обмениваться информацией с драйвером.

Единственное место где вызывается функция SSQC() это [thunk function](#):

```

.text:0000DBE8      public SSQ
.text:0000DBE8 SSQ   proc near ; CODE XREF: sys_info+A9p
.text:0000DBE8                ; sys_info+CBp ...
.text:0000DBE8 arg_0 = dword ptr 8
.text:0000DBE8      push    ebp
.text:0000DBE9      mov    ebp, esp
.text:0000DBEB      mov    edx, [ebp+arg_0]
.text:0000DBEE      push    edx
.text:0000DBEF      call   SSQC
.text:0000DBF4      add    esp, 4
.text:0000DBF7      mov    esp, ebp
.text:0000DBF9      pop    ebp
.text:0000DBFA      retn
.text:0000DBFB SSQ   endp

```

А вот SSQ() может вызываться по крайней мере из двух разных функций.

Одна из них:

```

.data:0040169C _51_52_53      dd offset aPressAnyKeyT_0 ;
                DATA XREF: init_sys+392r
.data:0040169C                ; sys_info+A1r
.data:0040169C                ; "PRESS ANY KEY TO
                CONTINUE: "
.data:004016A0                dd offset a51            ; "51"
.data:004016A4                dd offset a52            ; "52"
.data:004016A8                dd offset a53            ; "53"
...

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.data:004016B8 _3C_or_3E      dd offset a3c          ; DATA XREF:
                        sys_info:loc_D67Br
.data:004016B8              ; "3C"
.data:004016BC              dd offset a3e          ; "3E"

; эти имена мы сами дали этим меткам:
.data:004016C0 answers1    dd 6B05h              ; DATA XREF:
                        sys_info+E7r
.data:004016C4              dd 3D87h
.data:004016C8 answers2    dd 3Ch                ; DATA XREF:
                        sys_info+F2r
.data:004016CC              dd 832h
.data:004016D0 _C_and_B    db 0Ch                ; DATA XREF:
                        sys_info+BAr
.data:004016D0              ; sys_info:0Kr
.data:004016D1 byte_4016D1 db 0Bh                    ;
                        DATA XREF: sys_info+FDr
.data:004016D2              db 0

...

.text:0000D652             xor     eax, eax
.text:0000D654             mov     al, ds:ctl_port
.text:0000D659             mov     ecx, _51_52_53[eax*4]
.text:0000D660             push   ecx
.text:0000D661             call   SSQ
.text:0000D666             add     esp, 4
.text:0000D669             cmp     eax, 0FFFFFFFh
.text:0000D66E             jz     short loc_D6D1
.text:0000D670             xor     ebx, ebx
.text:0000D672             mov     al, _C_and_B
.text:0000D677             test   al, al
.text:0000D679             jz     short loc_D6C0
.text:0000D67B             loc_D67B: ; CODE XREF: sys_info+106j
.text:0000D67B             mov     eax, _3C_or_3E[ebx*4]
.text:0000D682             push   eax
.text:0000D683             call   SSQ
.text:0000D688             push   offset a4g      ; "4G"
.text:0000D68D             call   SSQ
.text:0000D692             push   offset a0123456789 ; "0123456789"
.text:0000D697             call   SSQ
.text:0000D69C             add     esp, 0Ch
.text:0000D69F             mov     edx, answers1[ebx*4]
.text:0000D6A6             cmp     eax, edx
.text:0000D6A8             jz     short 0K
.text:0000D6AA             mov     ecx, answers2[ebx*4]
.text:0000D6B1             cmp     eax, ecx
.text:0000D6B3             jz     short 0K
.text:0000D6B5             mov     al, byte_4016D1[ebx]
.text:0000D6BB             inc     ebx
.text:0000D6BC             test   al, al
.text:0000D6BE             jnz    short loc_D67B
.text:0000D6C0

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:0000D6C0 loc_D6C0: ; CODE XREF: sys_info+C1j
.text:0000D6C0         inc     ds:ctl_port
.text:0000D6C6         xor     eax, eax
.text:0000D6C8         mov     al, ds:ctl_port
.text:0000D6CD         cmp     eax, edi
.text:0000D6CF         jle     short loc_D652
.text:0000D6D1
.text:0000D6D1 loc_D6D1: ; CODE XREF: sys_info+98j
.text:0000D6D1         ; sys_info+B6j
.text:0000D6D1         mov     edx, [ebp+var_8]
.text:0000D6D4         inc     edx
.text:0000D6D5         mov     [ebp+var_8], edx
.text:0000D6D8         cmp     edx, 3
.text:0000D6DB         jle     loc_D641
.text:0000D6E1
.text:0000D6E1 loc_D6E1: ; CODE XREF: sys_info+16j
.text:0000D6E1         ; sys_info+51j ...
.text:0000D6E1         pop     ebx
.text:0000D6E2         pop     edi
.text:0000D6E3         mov     esp, ebp
.text:0000D6E5         pop     ebp
.text:0000D6E6         retn
.text:0000D6E8 OK:         ; CODE XREF: sys_info+F0j
.text:0000D6E8         ; sys_info+FBj
.text:0000D6E8         mov     al, _C_and_B[ebx]
.text:0000D6EE         pop     ebx
.text:0000D6EF         pop     edi
.text:0000D6F0         mov     ds:ctl_model, al
.text:0000D6F5         mov     esp, ebp
.text:0000D6F7         pop     ebp
.text:0000D6F8         retn
.text:0000D6F8 sys_info     endp

```

«3С» и «3Е» — это звучит знакомо: когда-то была донгла Sentinel Pro от Rainbow без памяти, предоставляющая только одну секретную крипто-хеширующую функцию.

О том, что такое хэш-функция, было описано здесь: ?? (стр. ??).

Но вернемся к нашей программе. Так что программа может только проверить подключена ли донгла или нет. Никакой больше информации в такую донглу без памяти записать нельзя. Двухсимвольные коды — это команды (можно увидеть, как они обрабатываются в функции SSQC ()) а все остальные строки хешируются внутри донглы превращаясь в 16-битное число. Алгоритм был секретный, так что нельзя было написать замену драйверу или сделать электронную копию донглы идеально эмулирующую алгоритм. С другой стороны, всегда можно было перехватить все обращения к ней и найти те константы, с которыми сравнивается результат хеширования. Но надо сказать, вполне возможно создать устойчивую защиту от копирования базирующуюся на секретной хеш-функции: пусть она шифрует все файлы с которыми ваша программа работает.

Но вернемся к нашему коду.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Коды 51/52/53 используются для выбора номера принтеровского LPT-порта. 3x/4x используются для выбора «family» так донглы Sentinel Pro можно отличать друг от друга: ведь более одной донглы может быть подключено к LPT-порту.

Единственная строка, передающаяся в хеш-функцию это "0123456789". Затем результат сравнивается с несколькими правильными значениями.

Если результат правилен, 0xC или 0xB будет записано в глобальную переменную `ctl_model`.

Еще одна строка для хеширования: "PRESS ANY KEY TO CONTINUE: ", но результат не проверяется. Трудно сказать, зачем это, может быть по ошибке <sup>16</sup>.

Давайте посмотрим, где проверяется значение глобальной переменной `ctl_model`.

Одно из таких мест:

```
.text:0000D708 prep_sys proc near ; CODE XREF: init_sys+46Ap
.text:0000D708
.text:0000D708 var_14 = dword ptr -14h
.text:0000D708 var_10 = byte ptr -10h
.text:0000D708 var_8 = dword ptr -8
.text:0000D708 var_2 = word ptr -2
.text:0000D708
.text:0000D708 push ebp
.text:0000D709 mov eax, ds:net_env
.text:0000D70E mov ebp, esp
.text:0000D710 sub esp, 1Ch
.text:0000D713 test eax, eax
.text:0000D715 jnz short loc_D734
.text:0000D717 mov al, ds:ctl_model
.text:0000D71C test al, al
.text:0000D71E jnz short loc_D77E
.text:0000D720 mov [ebp+var_8], offset aIeCvuInvv0kgT_ ;
    "Ie-cvuInvV\\b0KG]T_"
.text:0000D727 mov edx, 7
.text:0000D72C jmp loc_D7E7

...

.text:0000D7E7 loc_D7E7: ; CODE XREF: prep_sys+24j
.text:0000D7E7 ; prep_sys+33j
.text:0000D7E7 push edx
.text:0000D7E8 mov edx, [ebp+var_8]
.text:0000D7EB push 20h
.text:0000D7ED push edx
.text:0000D7EE push 16h
.text:0000D7F0 call err_warn
.text:0000D7F5 push offset station_sem
.text:0000D7FA call ClosSem
.text:0000D7FF call startup_err
```

<sup>16</sup>Это очень странное чувство: находить ошибки в столь древнем ПО.

Если оно 0, зашифрованное сообщение об ошибке будет передано в функцию дешифрования, и оно будет показано.

Функция дешифровки сообщений об ошибке похоже применяет простой [xor](#)ing:

```
.text:0000A43C err_warn      proc near          ; CODE XREF:
                    prep_sys+E8p
.text:0000A43C                                ; prep_sys2+2Fp ...
.text:0000A43C
.text:0000A43C var_55      = byte ptr -55h
.text:0000A43C var_54      = byte ptr -54h
.text:0000A43C arg_0       = dword ptr  8
.text:0000A43C arg_4       = dword ptr  0Ch
.text:0000A43C arg_8       = dword ptr  10h
.text:0000A43C arg_C       = dword ptr  14h
.text:0000A43C
.text:0000A43C          push    ebp
.text:0000A43D          mov     ebp, esp
.text:0000A43F          sub     esp, 54h
.text:0000A442          push    edi
.text:0000A443          mov     ecx, [ebp+arg_8]
.text:0000A446          xor     edi, edi
.text:0000A448          test   ecx, ecx
.text:0000A44A          push    esi
.text:0000A44B          jle    short loc_A466
.text:0000A44D          mov     esi, [ebp+arg_C] ; key
.text:0000A450          mov     edx, [ebp+arg_4] ; string
.text:0000A453
.text:0000A453 loc_A453:          ; CODE XREF:
                    err_warn+28j
.text:0000A453          xor     eax, eax
.text:0000A455          mov     al, [edx+edi]
.text:0000A458          xor     eax, esi
.text:0000A45A          add     esi, 3
.text:0000A45D          inc     edi
.text:0000A45E          cmp     edi, ecx
.text:0000A460          mov     [ebp+edi+var_55], al
.text:0000A464          jl     short loc_A453
.text:0000A466
.text:0000A466 loc_A466:          ; CODE XREF:
                    err_warn+Fj
.text:0000A466          mov     [ebp+edi+var_54], 0
.text:0000A46B          mov     eax, [ebp+arg_0]
.text:0000A46E          cmp     eax, 18h
.text:0000A473          jnz    short loc_A49C
.text:0000A475          lea    eax, [ebp+var_54]
.text:0000A478          push    eax
.text:0000A479          call   status_line
.text:0000A47E          add     esp, 4
.text:0000A481
.text:0000A481 loc_A481:          ; CODE XREF:
                    err_warn+72j
.text:0000A481          push    50h
.text:0000A483          push    0
.text:0000A485          lea    eax, [ebp+var_54]
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:0000A488      push     eax
.text:0000A489      call    memset
.text:0000A48E      call    pcv_refresh
.text:0000A493      add     esp, 0Ch
.text:0000A496      pop     esi
.text:0000A497      pop     edi
.text:0000A498      mov     esp, ebp
.text:0000A49A      pop     ebp
.text:0000A49B      retn
.text:0000A49C
.text:0000A49C loc_A49C:                                ; CODE XREF:
      err_warn+37j
.text:0000A49C      push    0
.text:0000A49E      lea    eax, [ebp+var_54]
.text:0000A4A1      mov     edx, [ebp+arg_0]
.text:0000A4A4      push   edx
.text:0000A4A5      push   eax
.text:0000A4A6      call   pcv_lputs
.text:0000A4AB      add     esp, 0Ch
.text:0000A4AE      jmp     short loc_A481
.text:0000A4AE err_warn      endp

```

Вот почему не получилось найти сообщение об ошибке в исполняемых файлах, потому что оно было зашифровано, это очень популярная практика.

Еще один вызов хеширующей функции передает строку «offln» и сравнивает результат с константами 0xFE81 и 0x12A9. Если результат не сходится, происходит работа с какой-то функцией timer() (может быть для ожидания плохо подключенной донглы и нового запроса?), затем дешифрует еще одно сообщение об ошибке и выводит его.

```

.text:0000DA55 loc_DA55:                                ; CODE XREF:
      sync_sys+24Cj
.text:0000DA55      push   offset aoffln ; "offln"
.text:0000DA5A      call   SSQ
.text:0000DA5F      add     esp, 4
.text:0000DA62      mov     dl, [ebx]
.text:0000DA64      mov     esi, eax
.text:0000DA66      cmp     dl, 0Bh
.text:0000DA69      jnz    short loc_DA83
.text:0000DA6B      cmp     esi, 0FE81h
.text:0000DA71      jz     OK
.text:0000DA77      cmp     esi, 0FFFFFF8EFh
.text:0000DA7D      jz     OK
.text:0000DA83
.text:0000DA83 loc_DA83:                                ; CODE XREF:
      sync_sys+201j
.text:0000DA83      mov     cl, [ebx]
.text:0000DA85      cmp     cl, 0Ch
.text:0000DA88      jnz    short loc_DA9F
.text:0000DA8A      cmp     esi, 12A9h
.text:0000DA90      jz     OK
.text:0000DA96      cmp     esi, 0FFFFFFF5h
.text:0000DA99      jz     OK

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:0000DA9F
.text:0000DA9F loc_DA9F: ; CODE XREF:
        sync_sys+220j
.text:0000DA9F      mov     eax, [ebp+var_18]
.text:0000DAA2      test    eax, eax
.text:0000DAA4      jz     short loc_DAB0
.text:0000DAA6      push   24h
.text:0000DAA8      call   timer
.text:0000DAAD      add    esp, 4
.text:0000DAB0
.text:0000DAB0 loc_DAB0: ; CODE XREF:
        sync_sys+23Cj
.text:0000DAB0      inc    edi
.text:0000DAB1      cmp    edi, 3
.text:0000DAB4      jle   short loc_DA55
.text:0000DAB6      mov    eax, ds:net_env
.text:0000DABB      test   eax, eax
.text:0000DABD      jz    short error
...

.text:0000DAF7 error: ; CODE XREF:
        sync_sys+255j
.text:0000DAF7 ; sync_sys+274j ...
.text:0000DAF7      mov    [ebp+var_8], offset ↙
        ↘ encrypted_error_message2
.text:0000DAFE      mov    [ebp+var_C], 17h ; decrypting key
.text:0000DB05      jmp   decrypt_end_print_message
...

; ЭТО ИМЯ МЫ САМИ ДАЛИ ЭТОЙ МЕТКЕ:
.text:0000D9B6 decrypt_end_print_message: ; CODE XREF:
        sync_sys+29Dj
.text:0000D9B6 ; sync_sys+2ABj
.text:0000D9B6      mov    eax, [ebp+var_18]
.text:0000D9B9      test   eax, eax
.text:0000D9BB      jnz   short loc_D9FB
.text:0000D9BD      mov    edx, [ebp+var_C] ; key
.text:0000D9C0      mov    ecx, [ebp+var_8] ; string
.text:0000D9C3      push   edx
.text:0000D9C4      push   20h
.text:0000D9C6      push   ecx
.text:0000D9C7      push   18h
.text:0000D9C9      call   err_warn
.text:0000D9CE      push   0Fh
.text:0000D9D0      push   190h
.text:0000D9D5      call   sound
.text:0000D9DA      mov    [ebp+var_18], 1
.text:0000D9E1      add    esp, 18h
.text:0000D9E4      call   pcv_kbhit
.text:0000D9E9      test   eax, eax
.text:0000D9EB      jz    short loc_D9FB

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

...
; это имя мы сами дали этой метке:
.data:00401736 encrypted_error_message2 db 74h, 72h, 78h, 43h, 48h, 6, 5Ah, ↵
    ↵ 49h, 4Ch, 2 dup(47h)
.data:00401736 db 51h, 4Fh, 47h, 61h, 20h, 22h, 3Ch, 24h, ↵
    ↵ 33h, 36h, 76h
.data:00401736 db 3Ah, 33h, 31h, 0Ch, 0, 0Bh, 1Fh, 7, 1Eh, ↵
    ↵ 1Ah

```

Заставить работать программу без донглы довольно просто: просто пропатчить все места после инструкций CMP где происходят соответствующие сравнения.

Еще одна возможность — это написать свой драйвер для SCO OpenServer, содержащий таблицу возможных вопросов и ответов, все те что имеются в программе.

### Дешифровка сообщений об ошибке

Кстати, мы также можем дешифровать все сообщения об ошибке. Алгоритм, находящийся в функции `err_warn()` действительно, крайне прост:

Листинг 8.5: Функция дешифровки

```

.text:0000A44D mov esi, [ebp+arg_C] ; key
.text:0000A450 mov edx, [ebp+arg_4] ; string
.text:0000A453 loc_A453:
.text:0000A453 xor eax, eax
.text:0000A455 mov al, [edx+edi] ; загружаем байт для
    дешифровки
.text:0000A458 xor eax, esi ; дешифуем его
.text:0000A45A add esi, 3 ; изменяем ключ для
    следующего байта
.text:0000A45D inc edi
.text:0000A45E cmp edi, ecx
.text:0000A460 mov [ebp+edi+var_55], al
.text:0000A464 jnl short loc_A453

```

Как видно, не только сама строка поступает на вход, но также и ключ для дешифровки:

```

.text:0000DAF7 error: ; CODE XREF:
    sync_sys+255j
.text:0000DAF7 ; sync_sys+274j ...
.text:0000DAF7 mov [ebp+var_8], offset ↵
    ↵ encrypted_error_message2
.text:0000DAFE mov [ebp+var_C], 17h ; decrypting key
.text:0000DB05 jmp decrypt_end_print_message
...
; это имя мы сами дали этой метке:

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

.text:0000D9B6 decrypt_end_print_message:          ; CODE XREF:
                sync_sys+29Dj
.text:0000D9B6                                ; sync_sys+2ABj
.text:0000D9B6      mov     eax, [ebp+var_18]
.text:0000D9B9      test    eax, eax
.text:0000D9BB      jnz    short loc_D9FB
.text:0000D9BD      mov     edx, [ebp+var_C] ; key
.text:0000D9C0      mov     ecx, [ebp+var_8] ; string
.text:0000D9C3      push   edx
.text:0000D9C4      push   20h
.text:0000D9C6      push   ecx
.text:0000D9C7      push   18h
.text:0000D9C9      call   err_warn

```

Алгоритм это очень простой [xoring](#): каждый байт XOR-ится с ключом, но ключ увеличивается на 3 после обработки каждого байта.

Напишем небольшой скрипт на Python для проверки наших догадок:

Листинг 8.6: Python 3.x

```

#!/usr/bin/python
import sys

msg=[0x74, 0x72, 0x78, 0x43, 0x48, 0x6, 0x5A, 0x49, 0x4C, 0x47, 0x47,
0x51, 0x4F, 0x47, 0x61, 0x20, 0x22, 0x3C, 0x24, 0x33, 0x36, 0x76,
0x3A, 0x33, 0x31, 0x0C, 0x0, 0x0B, 0x1F, 0x7, 0x1E, 0x1A]

key=0x17
tmp=key
for i in msg:
    sys.stdout.write ("%c" % (i^tmp))
    tmp=tmp+3
sys.stdout.flush()

```

И он выводит: «check security device connection». Так что да, это дешифрованное сообщение.

Здесь есть также и другие сообщения, с соответствующими ключами. Но надо сказать, их можно дешифровать и без ключей. В начале, мы можем заметить, что ключ — это просто байт. Это потому что самая важная часть функции дешифровки (XOR) оперирует байтами. Ключ находится в регистре ESI, но только младшие 8 бит (т.е. байт) регистра используются. Следовательно, ключ может быть больше 255, но его значение будет округляться.

И как следствие, мы можем попробовать обычный перебор всех ключей в диапазоне 0..255. Мы также можем пропускать сообщения содержащие непечатаемые символы.

Листинг 8.7: Python 3.x

```

#!/usr/bin/python
import sys, curses.ascii

msgs=[

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

[0x74, 0x72, 0x78, 0x43, 0x48, 0x6, 0x5A, 0x49, 0x4C, 0x47, 0x47,
0x51, 0x4F, 0x47, 0x61, 0x20, 0x22, 0x3C, 0x24, 0x33, 0x36, 0x76,
0x3A, 0x33, 0x31, 0x0C, 0x0, 0x0B, 0x1F, 0x7, 0x1E, 0x1A],

[0x49, 0x65, 0x2D, 0x63, 0x76, 0x75, 0x6C, 0x6E, 0x76, 0x56, 0x5C,
8, 0x4F, 0x4B, 0x47, 0x5D, 0x54, 0x5F, 0x1D, 0x26, 0x2C, 0x33,
0x27, 0x28, 0x6F, 0x72, 0x75, 0x78, 0x7B, 0x7E, 0x41, 0x44],

[0x45, 0x61, 0x31, 0x67, 0x72, 0x79, 0x68, 0x52, 0x4A, 0x52, 0x50,
0x0C, 0x4B, 0x57, 0x43, 0x51, 0x58, 0x5B, 0x61, 0x37, 0x33, 0x2B,
0x39, 0x39, 0x3C, 0x38, 0x79, 0x3A, 0x30, 0x17, 0x0B, 0x0C],

[0x40, 0x64, 0x79, 0x75, 0x7F, 0x6F, 0x0, 0x4C, 0x40, 0x9, 0x4D, 0x5A,
0x46, 0x5D, 0x57, 0x49, 0x57, 0x3B, 0x21, 0x23, 0x6A, 0x38, 0x23,
0x36, 0x24, 0x2A, 0x7C, 0x3A, 0x1A, 0x6, 0x0D, 0x0E, 0x0A, 0x14,
0x10],

[0x72, 0x7C, 0x72, 0x79, 0x76, 0x0,
0x50, 0x43, 0x4A, 0x59, 0x5D, 0x5B, 0x41, 0x41, 0x1B, 0x5A,
0x24, 0x32, 0x2E, 0x29, 0x28, 0x70, 0x20, 0x22, 0x38, 0x28, 0x36,
0x0D, 0x0B, 0x48, 0x4B, 0x4E]]

def is_string_printable(s):
    return all(list(map(lambda x: curses.ascii.isprint(x), s)))

cnt=1
for msg in msgs:
    print ("message #d" % cnt)
    for key in range(0,256):
        result=[]
        tmp=key
        for i in msg:
            result.append (i^tmp)
            tmp=tmp+3
        if is_string_printable (result):
            print ("key=", key, "value=", "".join(list(map(chr,
↵ result))))
            cnt=cnt+1

```

И мы получим:

#### Листинг 8.8: Results

```

message #1
key= 20 value= `eb^h%|\`hudw|_af{n~f%ljmSbnwlpk
key= 21 value= ajc|i"}cawtg{^bgto}g"millcmvkqh
key= 22 value= bkd\j#rbbvsfuz!cduh|d#bhomdlujni
key= 23 value= check security device connection
key= 24 value= lifbl!pd|tqhsx#ejwbbb!\nQofbshlo
message #2
key= 7 value= No security device found
key= 8 value= An#rbbvsVuz!cduhld#ghtme?!#!'#!
message #3

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

key= 7 value= Bk<waoqNUpu$`yreaa\wmpusj,bkIjh
key= 8 value= Mj?vfnr0jqv%gxqd``_vwlstlk/clHii
key= 9 value= Lm>ugasLkvw&fgpgag^uvcrwml.`mwhj
key= 10 value= 0l!td`tmhwx'efwfbf!tubuvnm!anvok
key= 11 value= No security device station found
key= 12 value= In#rjbvsnuz!{duhdd#r{`whho#gPtme
message #4
key= 14 value= Number of authorized users exceeded
key= 15 value= 0vlmdq!hg#`juknuhydk!vrbsp!Zy`dbefe
message #5
key= 17 value= check security device station
key= 18 value= `ijbh!td`tmhwx'efwfbf!tubuVnm!'!

```

Тут есть какой-то мусор, но мы можем быстро отыскать сообщения на английском языке!

Кстати, так как алгоритм использует простой XOR, та же функция может использоваться и для шифрования сообщения. Если нужно, мы можем зашифровать наши собственные сообщения, и пропатчить программу вставив их.

### 8.6.3. Пример #3: MS-DOS

Еще одна очень старая программа для MS-DOS от 1995 также разработанная давно исчезнувшей компанией.

Во времена перед DOS-экстендерами, всё ПО для MS-DOS рассчитывалось на процессоры 8086 или 80286, так что в своей массе весь код был 16-битным. 16-битный код в основном такой же, какой вы уже видели в этой книге, но все регистры 16-битные, и доступно меньше инструкций.

Среда MS-DOS не могла иметь никаких драйверов, и ПО работало с «голым» железом через порты, так что здесь вы можете увидеть инструкции OUT/IN, которые в наше время присутствуют в основном только в драйверах (в современных OS нельзя обращаться напрямую к портам из [user mode](#)).

Учитывая это, ПО для MS-DOS должно работать с донглой обращаясь к принтерному LPT-порту напрямую. Так что мы можем просто поискать эти инструкции. И да, вот они:

```

seg030:0034          out_port proc far ; CODE XREF: sent_pro+22p
seg030:0034                               ; sent_pro+2Ap ...
seg030:0034
seg030:0034          arg_0      = byte ptr 6
seg030:0034
seg030:0034 55             push     bp
seg030:0035 8B EC          mov     bp, sp
seg030:0037 8B 16 7E E7    mov     dx, _out_port ; 0x378
seg030:003B 8A 46 06       mov     al, [bp+arg_0]
seg030:003E EE             out     dx, al
seg030:003F 5D             pop     bp
seg030:0040 CB             retf
seg030:0040          out_port endp

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

(Все имена меток в этом примере даны мною).

Функция out\_port() вызывается только из одной функции:

seg030:0041		sent_pro proc far ; CODE XREF: check_dongle+34p
seg030:0041		
seg030:0041		var_3 = byte ptr -3
seg030:0041		var_2 = word ptr -2
seg030:0041		arg_0 = dword ptr 6
seg030:0041		
seg030:0041	C8 04 00 00	enter 4, 0
seg030:0045	56	push si
seg030:0046	57	push di
seg030:0047	8B 16 82 E7	mov dx, _in_port_1 ; 0x37A
seg030:004B	EC	in al, dx
seg030:004C	8A D8	mov bl, al
seg030:004E	80 E3 FE	and bl, 0FEh
seg030:0051	80 CB 04	or bl, 4
seg030:0054	8A C3	mov al, bl
seg030:0056	88 46 FD	mov [bp+var_3], al
seg030:0059	80 E3 1F	and bl, 1Fh
seg030:005C	8A C3	mov al, bl
seg030:005E	EE	out dx, al
seg030:005F	68 FF 00	push 0FFh
seg030:0062	0E	push cs
seg030:0063	E8 CE FF	call near ptr out_port
seg030:0066	59	pop cx
seg030:0067	68 D3 00	push 0D3h
seg030:006A	0E	push cs
seg030:006B	E8 C6 FF	call near ptr out_port
seg030:006E	59	pop cx
seg030:006F	33 F6	xor si, si
seg030:0071	EB 01	jmp short loc_359D4
seg030:0073		
seg030:0073		loc_359D3: ; CODE XREF: sent_pro+37j
seg030:0073	46	inc si
seg030:0074		
seg030:0074		loc_359D4: ; CODE XREF: sent_pro+30j
seg030:0074	81 FE 96 00	cmp si, 96h
seg030:0078	7C F9	jl short loc_359D3
seg030:007A	68 C3 00	push 0C3h
seg030:007D	0E	push cs
seg030:007E	E8 B3 FF	call near ptr out_port
seg030:0081	59	pop cx
seg030:0082	68 C7 00	push 0C7h
seg030:0085	0E	push cs
seg030:0086	E8 AB FF	call near ptr out_port
seg030:0089	59	pop cx
seg030:008A	68 D3 00	push 0D3h
seg030:008D	0E	push cs
seg030:008E	E8 A3 FF	call near ptr out_port
seg030:0091	59	pop cx
seg030:0092	68 C3 00	push 0C3h
seg030:0095	0E	push cs

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

seg030:0096 E8 9B FF      call    near ptr out_port
seg030:0099 59              pop     cx
seg030:009A 68 C7 00      push   0C7h
seg030:009D 0E              push   cs
seg030:009E E8 93 FF      call    near ptr out_port
seg030:00A1 59              pop     cx
seg030:00A2 68 D3 00      push   0D3h
seg030:00A5 0E              push   cs
seg030:00A6 E8 8B FF      call    near ptr out_port
seg030:00A9 59              pop     cx
seg030:00AA BF FF FF      mov     di, 0FFFFh
seg030:00AD EB 40          jmp     short loc_35A4F
seg030:00AF
seg030:00AF      loc_35A0F: ; CODE XREF: sent_pro+BDj
seg030:00AF BE 04 00      mov     si, 4
seg030:00B2
seg030:00B2      loc_35A12: ; CODE XREF: sent_pro+ACj
seg030:00B2 D1 E7          shl     di, 1
seg030:00B4 8B 16 80 E7    mov     dx, _in_port_2 ; 0x379
seg030:00B8 EC              in      al, dx
seg030:00B9 A8 80          test    al, 80h
seg030:00BB 75 03          jnz     short loc_35A20
seg030:00BD 83 CF 01      or      di, 1
seg030:00C0
seg030:00C0      loc_35A20: ; CODE XREF: sent_pro+7Aj
seg030:00C0 F7 46 FE 08+   test    [bp+var_2], 8
seg030:00C5 74 05          jz      short loc_35A2C
seg030:00C7 68 D7 00      push   0D7h ; '+'
seg030:00CA EB 0B          jmp     short loc_35A37
seg030:00CC
seg030:00CC      loc_35A2C: ; CODE XREF: sent_pro+84j
seg030:00CC 68 C3 00      push   0C3h
seg030:00CF 0E              push   cs
seg030:00D0 E8 61 FF      call    near ptr out_port
seg030:00D3 59              pop     cx
seg030:00D4 68 C7 00      push   0C7h
seg030:00D7
seg030:00D7      loc_35A37: ; CODE XREF: sent_pro+89j
seg030:00D7 0E              push   cs
seg030:00D8 E8 59 FF      call    near ptr out_port
seg030:00DB 59              pop     cx
seg030:00DC 68 D3 00      push   0D3h
seg030:00DF 0E              push   cs
seg030:00E0 E8 51 FF      call    near ptr out_port
seg030:00E3 59              pop     cx
seg030:00E4 8B 46 FE      mov     ax, [bp+var_2]
seg030:00E7 D1 E0          shl     ax, 1
seg030:00E9 89 46 FE      mov     [bp+var_2], ax
seg030:00EC 4E              dec     si
seg030:00ED 75 C3          jnz     short loc_35A12
seg030:00EF
seg030:00EF      loc_35A4F: ; CODE XREF: sent_pro+6Cj
seg030:00EF C4 5E 06      les     bx, [bp+arg_0]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

seg030:00F2 FF 46 06      inc     word ptr [bp+arg_0]
seg030:00F5 26 8A 07      mov     al, es:[bx]
seg030:00F8 98           cbw
seg030:00F9 89 46 FE      mov     [bp+var_2], ax
seg030:00FC 0B C0        or      ax, ax
seg030:00FE 75 AF        jnz     short loc_35A0F
seg030:0100 68 FF 00     push   0FFh
seg030:0103 0E           push   cs
seg030:0104 E8 2D FF     call   near ptr out_port
seg030:0107 59           pop     cx
seg030:0108 8B 16 82 E7  mov     dx, _in_port_1 ; 0x37A
seg030:010C EC           in      al, dx
seg030:010D 8A C8        mov     cl, al
seg030:010F 80 E1 5F     and     cl, 5Fh
seg030:0112 8A C1        mov     al, cl
seg030:0114 EE           out     dx, al
seg030:0115 EC           in      al, dx
seg030:0116 8A C8        mov     cl, al
seg030:0118 F6 C1 20     test    cl, 20h
seg030:011B 74 08        jz      short loc_35A85
seg030:011D 8A 5E FD     mov     bl, [bp+var_3]
seg030:0120 80 E3 DF     and     bl, 0DFh
seg030:0123 EB 03        jmp     short loc_35A88
seg030:0125
loc_35A85: ; CODE XREF: sent_pro+DAj
seg030:0125 8A 5E FD     mov     bl, [bp+var_3]
seg030:0128
loc_35A88: ; CODE XREF: sent_pro+E2j
seg030:0128 F6 C1 80     test    cl, 80h
seg030:012B 74 03        jz      short loc_35A90
seg030:012D 80 E3 7F     and     bl, 7Fh
seg030:0130
loc_35A90: ; CODE XREF: sent_pro+EAj
seg030:0130 8B 16 82 E7  mov     dx, _in_port_1 ; 0x37A
seg030:0134 8A C3        mov     al, bl
seg030:0136 EE           out     dx, al
seg030:0137 8B C7        mov     ax, di
seg030:0139 5F           pop     di
seg030:013A 5E           pop     si
seg030:013B C9           leave
seg030:013C CB           retf
seg030:013C      sent_pro endp

```

Это также «хеширующая» донгла Sentinel Pro как и в предыдущем примере. Это заметно по тому что текстовые строки передаются и здесь, 16-битные значения также возвращаются и сравниваются с другими.

Так вот как происходит работа с Sentinel Pro через порты. Адрес выходного порта обычно 0x378, т.е. принтерного порта, данные для него во времена перед USB отправлялись прямо сюда. Порт однонаправленный, потому что когда его разрабатывали, никто не мог предположить, что кому-то понадобится по-

лучать информацию из принтера <sup>17</sup>. Единственный способ получить информацию из принтера это регистр статуса на порту 0x379, он содержит такие биты как «paper out», «ask», «busy» — так принтер может сигнализировать о том, что он готов или нет, и о том, есть ли в нем бумага. Так что донгла возвращает информацию через какой-то из этих бит, по одному биту на каждой итерации.

`_in_port_2` содержит адрес статуса (0x379) и `_in_port_1` содержит адрес управляющего регистра (0x37A).

Судя по всему, донгла возвращает информацию только через флаг «busy» на `seg030:00B9`: каждый бит записывается в регистре DI позже возвращаемый в самом конце функции.

Что означают все эти отсылаемые в выходной порт байты? Трудно сказать. Возможно, команды донглы. Но честно говоря, нам и не обязательно знать: нашу задачу можно легко решить и без этих знаний.

Вот функция проверки донглы:

```

00000000 struct_0      struc ; (sizeof=0x1B)
00000000 field_0     db 25 dup(?)           ; string(C)
00000019 _A         dw ?
0000001B struct_0     ends

dseg:3CBC 61 63 72 75+_Q struct_0 <'hello', 01122h>
dseg:3CBC 6E 00 00 00+   ; DATA XREF: check_dongle+2E0

... skipped ...

dseg:3E00 63 6F 66 66+   struct_0 <'coffee', 7EB7h>
dseg:3E1B 64 6F 67 00+   struct_0 <'dog', 0FFADh>
dseg:3E36 63 61 74 00+   struct_0 <'cat', 0FF5Fh>
dseg:3E51 70 61 70 65+   struct_0 <'paper', 0FFDFh>
dseg:3E6C 63 6F 6B 65+   struct_0 <'coke', 0F568h>
dseg:3E87 63 6C 6F 63+   struct_0 <'clock', 55EAh>
dseg:3EA2 64 69 72 00+   struct_0 <'dir', 0FFAEh>
dseg:3EBD 63 6F 70 79+   struct_0 <'copy', 0F557h>

seg030:0145          check_dongle proc far ; CODE XREF: sub_3771D+3EP
seg030:0145
seg030:0145          var_6 = dword ptr -6
seg030:0145          var_2 = word ptr -2
seg030:0145
seg030:0145 C8 06 00 00          enter    6, 0
seg030:0149 56                push    si
seg030:014A 66 6A 00          push    large 0           ; newtime
seg030:014D 6A 00            push    0                 ; cmd
seg030:014F 9A C1 18 00+      call    _biostime
seg030:0154 52                push    dx
seg030:0155 50                push    ax
seg030:0156 66 58            pop     eax

```

<sup>17</sup>Если учитывать только Centronics и не учитывать последующий стандарт IEEE 1284 — в нем из принтера можно получать информацию.

seg030:0158	83 C4 06	add	sp, 6
seg030:015B	66 89 46 FA	mov	[bp+var_6], eax
seg030:015F	66 3B 06 D8+	cmp	eax, _expiration
seg030:0164	7E 44	jle	short loc_35B0A
seg030:0166	6A 14	push	14h
seg030:0168	90	nop	
seg030:0169	0E	push	cs
seg030:016A	E8 52 00	call	near ptr get_rand
seg030:016D	59	pop	cx
seg030:016E	8B F0	mov	si, ax
seg030:0170	6B C0 1B	imul	ax, 1Bh
seg030:0173	05 BC 3C	add	ax, offset _Q
seg030:0176	1E	push	ds
seg030:0177	50	push	ax
seg030:0178	0E	push	cs
seg030:0179	E8 C5 FE	call	near ptr sent_pro
seg030:017C	83 C4 04	add	sp, 4
seg030:017F	89 46 FE	mov	[bp+var_2], ax
seg030:0182	8B C6	mov	ax, si
seg030:0184	6B C0 12	imul	ax, 18
seg030:0187	66 0F BF C0	movsx	eax, ax
seg030:018B	66 8B 56 FA	mov	edx, [bp+var_6]
seg030:018F	66 03 D0	add	edx, eax
seg030:0192	66 89 16 D8+	mov	_expiration, edx
seg030:0197	8B DE	mov	bx, si
seg030:0199	6B DB 1B	imul	bx, 27
seg030:019C	8B 87 D5 3C	mov	ax, _Q._A[bx]
seg030:01A0	3B 46 FE	cmp	ax, [bp+var_2]
seg030:01A3	74 05	jz	short loc_35B0A
seg030:01A5	B8 01 00	mov	ax, 1
seg030:01A8	EB 02	jmp	short loc_35B0C
seg030:01AA			
seg030:01AA		loc_35B0A:	; CODE XREF: check_dongle+1Fj
seg030:01AA			; check_dongle+5Ej
seg030:01AA	33 C0	xor	ax, ax
seg030:01AC			
seg030:01AC		loc_35B0C:	; CODE XREF: check_dongle+63j
seg030:01AC	5E	pop	si
seg030:01AD	C9	leave	
seg030:01AE	CB	retf	
seg030:01AE		check_dongle	endp

А так как эта функция может вызываться слишком часто, например, перед выполнением каждой важной возможности ПО, а обращение к донгле вообще-то медленное (и из-за медленного принтерного порта, и из-за медленного **MCU!** в донгле), так что они, наверное, добавили возможность пропускать проверку донглы слишком часто, используя текущее время в функции `biostime()`.

Функция `get_rand()` использует стандартную функцию Си:

seg030:01BF		get_rand	proc far ; CODE XREF: check_dongle+25p
seg030:01BF			
seg030:01BF		arg_0	= word ptr 6
seg030:01BF			

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

seg030:01BF 55          push    bp
seg030:01C0 8B EC          mov     bp, sp
seg030:01C2 9A 3D 21 00+   call   _rand
seg030:01C7 66 0F BF C0    movsx  eax, ax
seg030:01CB 66 0F BF 56+   movsx  edx, [bp+arg_0]
seg030:01D0 66 0F AF C2    imul   eax, edx
seg030:01D4 66 BB 00 80+   mov     ebx, 8000h
seg030:01DA 66 99          cdq
seg030:01DC 66 F7 FB      idiv   ebx
seg030:01DF 5D             pop     bp
seg030:01E0 CB             retf
seg030:01E0          get_rand endp

```

Так что текстовая строка выбирается случайно, отправляется в донглу и результат хеширования сверяется с корректным значением.

Текстовые строки, похоже, составлялись так же случайно, во время разработки ПО.

И вот как вызывается главная процедура проверки донглы:

```

seg033:087B 9A 45 01 96+   call   check_dongle
seg033:0880 0B C0          or     ax, ax
seg033:0882 74 62          jz     short OK
seg033:0884 83 3E 60 42+   cmp    word_620E0, 0
seg033:0889 75 5B          jnz    short OK
seg033:088B FF 06 60 42    inc    word_620E0
seg033:088F 1E             push   ds
seg033:0890 68 22 44      push   offset aTrupcRequiresA ;
"This Software Requires a Software Lock\n"
seg033:0893 1E             push   ds
seg033:0894 68 60 E9      push   offset byte_6C7E0 ; dest
seg033:0897 9A 79 65 00+   call   _strcpy
seg033:089C 83 C4 08      add    sp, 8
seg033:089F 1E             push   ds
seg033:08A0 68 42 44      push   offset aPleaseContactA ; "Please Contact
"
seg033:08A3 1E             push   ds
seg033:08A4 68 60 E9      push   offset byte_6C7E0 ; dest
seg033:08A7 9A CD 64 00+   call   _strcat

```

Заставить работать программу без донглы очень просто: просто заставить функцию `check_dongle()` возвращать всегда 0.

Например, вставив такой код в самом её начале:

```

mov ax,0
retf

```

Наблюдательный читатель может заметить, что функция Си `strcpy()` имеет 2 аргумента, но здесь мы видим, что передается 4:

```

seg033:088F 1E             push   ds
seg033:0890 68 22 44      push   offset aTrupcRequiresA ;
"This Software Requires a Software Lock\n"

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

seg033:0893 1E	push	ds
seg033:0894 68 60 E9	push	offset byte_6C7E0 ; dest
seg033:0897 9A 79 65 00+	call	_strcpy
seg033:089C 83 C4 08	add	sp, 8

Это связано с моделью памяти в MS-DOS. Об этом больше читайте здесь: ?? (стр. ??).

Так что, `strcpy()`, как и любая другая функция принимающая указатель (-и) в аргументах, работает с 16-битными парами.

Вернемся к нашему примеру. DS сейчас указывает на сегмент данных размещенный в исполняемом файле, там, где хранится текстовая строка.

В функции `sent_pro()` каждый байт строки загружается на `seg030:00EF`: инструкция `LES` загружает из переданного аргумента пару `ES:BX` одновременно. `MOV` на `seg030:00F5` загружает байт из памяти, на который указывает пара `ES:BX`.

## 8.7. Случай с зашифрованной БД #1

(Эта часть впервые появилась в моем блоге 26-Aug-2015. Обсуждение: <https://news.ycombinator.com/item?id=10128684>.)

### 8.7.1. Base64 и энтропия

Мне достался **XML!**-файл, содержащий некоторые зашифрованные данные. Вероятно, что-то связанное с заказами и/или с информацией о клиентах.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<Orders>
  <Order>
    <OrderID>1</OrderID>
    <Data>yjmxhXUhbB/5MV45chPsXZWAJwIh1S0aD9lFn3XuJMSxJ3/E+↵
  ↵ UE3hsnH</Data>
  </Order>
  <Order>
    <OrderID>2</OrderID>
    <Data>0KGe/wnypFBjsy+U0C2P9fC5nDZP3XDZLMPC RaiBw90jIk6Tu5U↵
  ↵ =</Data>
  </Order>
  <Order>
    <OrderID>3</OrderID>
    <Data>mqkXfdzvQKvEArdzh+zD9oETVGBFvcTBLs2ph1b5bYddExzp</↵
  ↵ Data>
  </Order>
  <Order>
    <OrderID>4</OrderID>
    <Data>FCx6JhIDqnESyT3HAepyE1BJ3cJd7wCk+APCRUeuNtZdpCvQ2MR/7↵
  ↵ kLXtFUHuA==</Data>
  </Order>
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

...

Файл доступен [здесь](#).

Это явно данные закодированные в base64, потому что все строки состоят из латинских символов, цифр, и символов плюс (+) и слэш (/). Могут быть еще два выравнивающих символа (=), но они никогда не встречаются в середине строки. Зная эти свойства base64, такие строки легко распознавать.

Попробуем декодировать эти блоки и вычислить их энтропии (?? (стр. ??)) при помощи Wolfram Mathematica:

```
In[ ]:= ListOfBase64Strings =
  Map[First#[[3]] &, Cases[Import["encrypted.xml"], XMLElement["Data", _, ↵
    ↵ _], Infinity]];

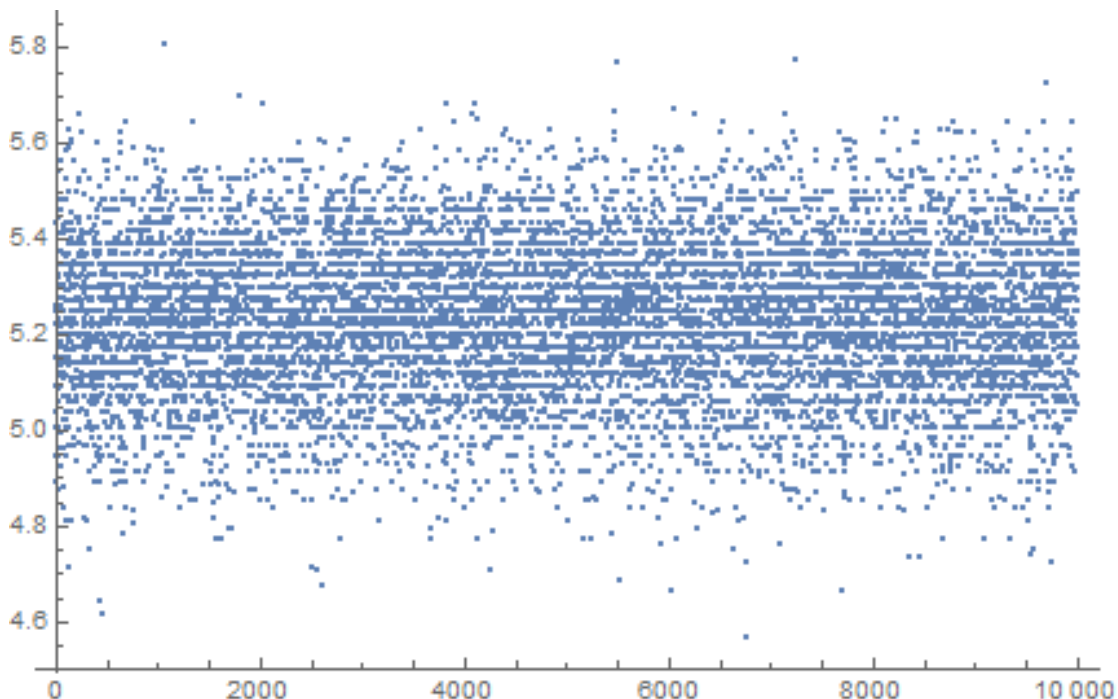
In[ ]:= BinaryStrings =
  Map[ImportString[#, {"Base64", "String"}] &, ListOfBase64Strings];

In[ ]:= Entropies = Map[N[Entropy[2, #]] &, BinaryStrings];

In[ ]:= Variance[Entropies]
Out[ ]= 0.0238614
```

Разброс (variance) низкий. Это означает, что значения энтропии не очень отличаются друг от друга. Это видно на графике:

```
In[ ]:= ListPlot[Entropies]
```

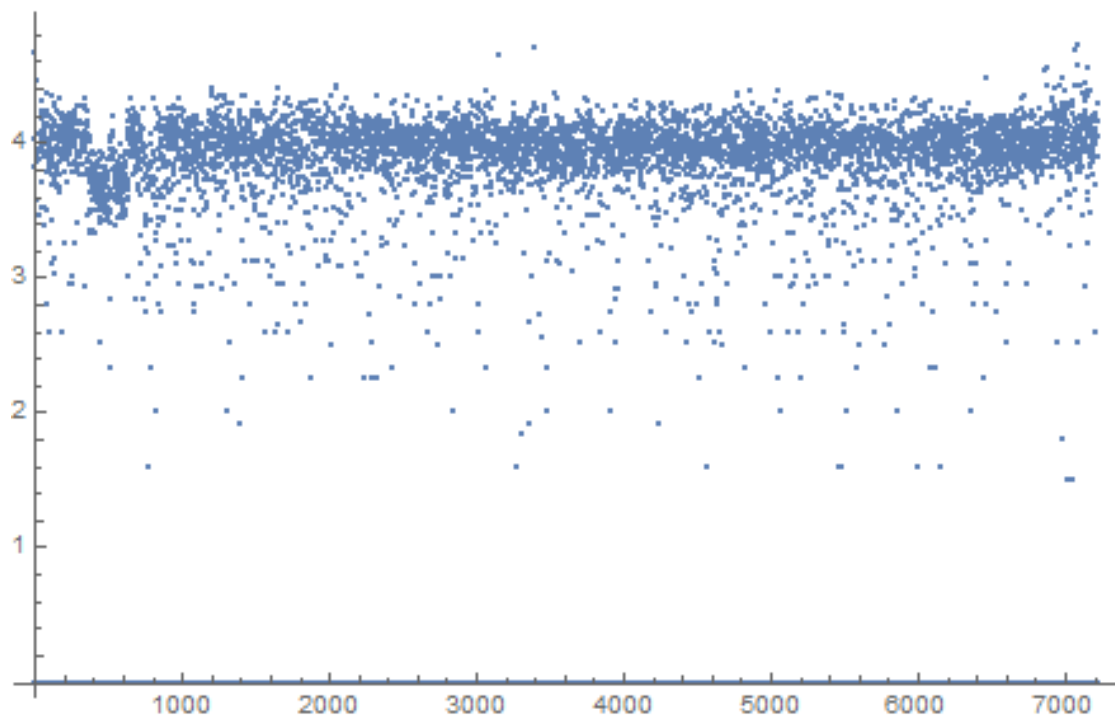


Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Большинство значений между 5.0 и 5.4. Это свидетельство того что данные сжаты и/или зашифрованы.

Чтобы понять разброс (variance), подсчитаем энтропии всех строк в книге Конана Дойля *The Hound of the Baskervilles*:

```
In[]:= BaskervillesLines = Import["http://www.gutenberg.org/cache/epub/2852/pg2852.txt", "List"];
In[]:= EntropiesT = Map[N[Entropy[2, #]] &, BaskervillesLines];
In[]:= Variance[EntropiesT]
Out[]= 2.73883
In[]:= ListPlot[EntropiesT]
```



Большинство значений находится вокруг 4, но есть также меньшие значения, и они повлияли на конечное значение разброса.

Вероятно, самые короткие строки имеют меньшую энтропию, попробуем короткую строку из книги Конан Дойля:

```
In[]:= Entropy[2, "Yes, sir."] // N
Out[]= 2.9477
```

Попробуем еще меньшую:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
In[]:= Entropy[2, "Yes"] // N
Out[]= 1.58496

In[]:= Entropy[2, "No"] // N
Out[]= 1.
```

### 8.7.2. Данные сжаты?

ОК, наши данные сжаты и/или зашифрованы. Сжаты ли? Почти все компрессоры данных помещают некоторый заголовок в начале, сигнатуру или что-то вроде этого. Как видим, здесь ничего такого нет в начале каждого блока. Все еще возможно что это какой-то самодельный компрессор, но они очень редки. С другой стороны, самодельные криптоалгоритмы попадают часто, потому что их куда легче заставить работать. Даже примитивные криптосистемы без ключей, как *memfrob()*<sup>18</sup> и ROT13 нормально работают без ошибок. А чтобы написать свой компрессор с нуля, используя только фантазию и воображение, так что он будет работать без ошибок, это серьезная задача. Некоторые программисты реализуют ф-ции сжатия данных по учебникам, но это также редкость. Наиболее популярные способы это: 1) просто взять опен-сорсную библиотеку вроде zlib; 2) скопипастить что-то откуда-то. Опен-сорсные алгоритмы сжатия данных обычно добавляют какой-то заголовок, и точно так же делают алгоритмы с сайтов вроде <http://www.codeproject.com/>.

### 8.7.3. Данные зашифрованы?

Основные алгоритмы шифрования обрабатывают данные блоками. DES — по 8 байт, AES — по 16 байт. Если входной буфер не делится без остатка на длину блока, он дополняется нулями (или еще чем-то), так что зашифрованные данные будут выровнены по размеру блока этого алгоритма шифрования. Это не наш случай.

Используя Wolfram Mathematica, я проанализировал длины блоков:

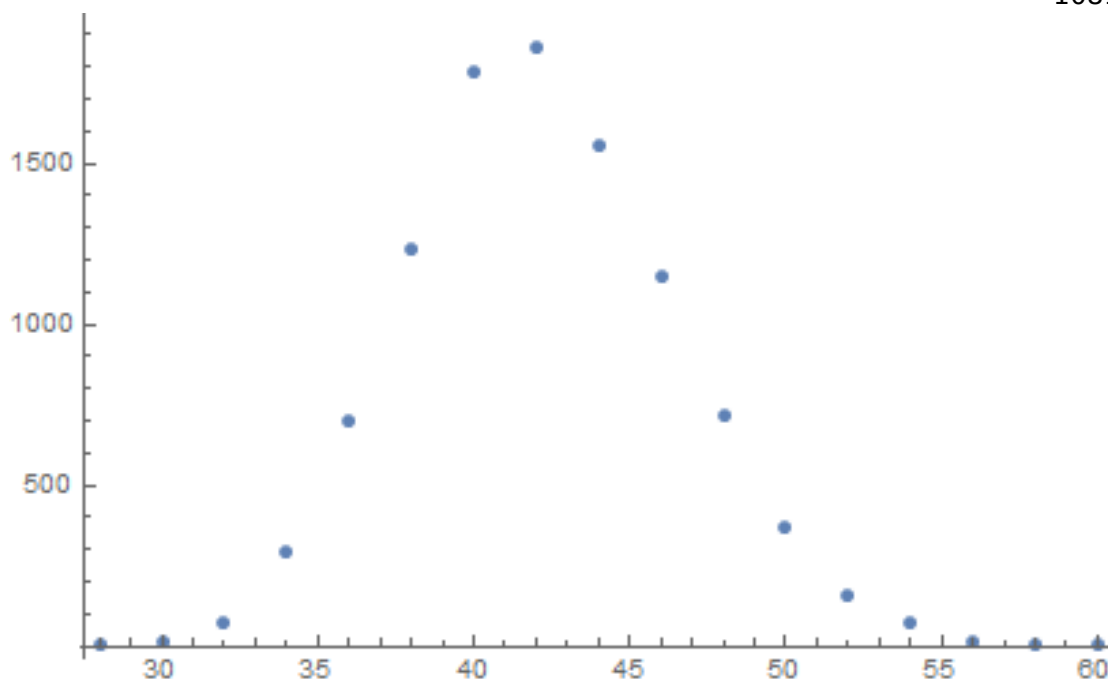
```
In[]:= Counts[Map[StringLength[#] &, BinaryStrings]]
Out[]= <|42 -> 1858, 38 -> 1235, 36 -> 699, 46 -> 1151, 40 -> 1784,
  44 -> 1558, 50 -> 366, 34 -> 291, 32 -> 74, 56 -> 15, 48 -> 716,
  30 -> 13, 52 -> 156, 54 -> 71, 60 -> 3, 58 -> 6, 28 -> 4|>
```

1858 блоков имеют длину 42 байта, 1235 блоков имеют длину 38 байт, и т. д.

Я сделал график:

```
ListPlot[Counts[Map[StringLength[#] &, BinaryStrings]]]
```

<sup>18</sup><http://linux.die.net/man/3/memfrob>



Так что большинство блоков имеют размер между  $\sim 36$  и  $\sim 48$ . Вот еще что стоит отметить: длины всех блоков четные. Нет ни одного блока с нечетной длиной.

Хотя, существуют потоковые шифры, которые работают на уровне байт, или даже на уровне бит.

#### 8.7.4. CryptoPP

Программа, при помощи которой можно листать зашифрованную базу написана на C# и код на .NET сильно обфусцирован. Тем не менее, имеется DLL с кодом для x86, который, после краткого рассмотрения, имеет части из популярной опен-сорсной библиотеки CryptoPP! (Я просто нашел внутри строки «CryptoPP».) Теперь легко найти все ф-ции внутри DLL, потому что библиотека CryptoPP опен-сорсная.

Библиотека CryptoPP имеет множество ф-ций шифрования, включая AES (АКА Rijndael). Современные x86-процессоры имеют AES-инструкции вроде AESEN, AESDEC и AESKEYGENASSIST <sup>19</sup>. Они не производят полного шифрования/дешифрования, но они делают бóльшую часть работы. И новые версии CryptoPP используют их. Например, здесь: [1](#), [2](#). К моему удивлению, во время дешифрования, инструкция, AESEN выполняется, а AESDEC — нет (я это проверил при помощи моей утилиты tracer, но можно использовать любой отладчик). Я проверил, поддерживает ли мой процессор AES-инструкции. Некоторые процессоры Intel i3 не поддерживают. И если нет, библиотека CryptoPP применяет ф-ции AES re-

<sup>19</sup>[https://en.wikipedia.org/wiki/AES\\_instruction\\_set](https://en.wikipedia.org/wiki/AES_instruction_set)

ализованные старым способом <sup>20</sup>. Но мой процессор поддерживает их. Почему AESDEC не исполняется? Почему программа использует шифрование AES чтобы дешифровать БД?

ОК, найти ф-цию шифрования блока это не проблема. Она называется `CryptoPP::Rijndael::Enc::ProcessAndXorBlock`: [src](#), и она может вызывать другую ф-цию:

`Rijndael::Enc::AdvancedProcessBlocks()` [src](#), которая, в свою очередь, может вызывать две ф-ции ( `AESNI_Enc_Block` and `AESNI_Enc_4_Blocks` ) которые имеют инструкции AESENC.

Так что, судя по внутренностям `CryptoPP`, `CryptoPP::Rijndael::Enc::ProcessAndXorBlock()` шифрует один 16-байтный блок. Попробуем установить брякпоинт на ней и посмотрим, что происходит во время дешифрования. Я снова использую мою простую утилиту `tracer`. Сейчас программа должна дешифровать первый блок. О, и кстати, вот первый блок сконвертированный из кодировки `base64` в шестнадцатеричный вид, будем держать его под рукой:

```
00000000: CA 39 B1 85 75 1B 84 1F F9 31 5E 39 72 13 EC 5D .9..u....1^9r↵
    ↵ ..]
00000010: 95 80 27 02 21 D5 2D 1A 0F D9 45 9F 75 EE 24 C4 ..'!.-...E.u.$↵
    ↵ .
00000020: B1 27 7F 84 FE 41 37 86 C9 C0 .'...A7...
```

А еще вот аргументы ф-ции из исходных файлов `CryptoPP`:

```
size_t Rijndael::Enc::AdvancedProcessBlocks(const byte *inBlocks, const ↵
    ↵ byte *xorBlocks, byte *outBlocks, size_t length, word32 flags);
```

Так что у него 5 аргументов. Возможные флаги это:

```
enum {BT_InBlockIsCounter=1, BT_DontIncrementInOutPointers=2, BT_XorInput↵
    ↵ =4, BT_ReverseDirection=8, BT_AllowParallel=16} ↵
    ↵ FlagsForAdvancedProcessBlocks;
```

ОК, запускаем `tracer` на ф-ции `ProcessAndXorBlock()`:

```
... tracer.exe -l:filename.exe bpf=filename.exe!0x4339a0,args:5,dump_args:0↵
    ↵ x10

Warning: no tracer.cfg file.
PID=1984|New process software.exe
no module registered with image base 0x77320000
no module registered with image base 0x76e20000
no module registered with image base 0x77320000
no module registered with image base 0x77220000
Warning: unknown (to us) INT3 breakpoint at ntdll.dll!↵
    ↵ LdrVerifyImageMatchesChecksum+0x96c (0x776c103b)
(0) software.exe!0x4339a0(0x38b920, 0x0, 0x38b978, 0x10, 0x0) (called from ↵
    ↵ software.exe!.text+0x33c0d (0x13e4c0d))
```

<sup>20</sup><https://github.com/mmooss/cryptopp/blob/2772f7b57182b31a41659b48d5f35a7b6cedd34d/src/rijndael.cpp#L355>

```

Argument 1/5
0038B920: 01 00 00 00 FF FF FF FF-79 C1 69 0B 67 C1 04 7D ". . . . . . . . . . y.i.g
↳ ..}"
Argument 3/5
0038B978: CD CD CD CD CD CD CD CD-CD CD CD CD CD CD CD CD
↳ " . . . . . . . . . . ."
(0) software.exe!0x4339a0() -> 0x0
Argument 3/5 difference
00000000: C7 39 4E 7B 33 1B D6 1F-B8 31 10 39 39 13 A5 5D ".9N
↳ {3....1.99..}"
(0) software.exe!0x4339a0(0x38a828, 0x38a838, 0x38bb40, 0x0, 0x8) (called
↳ from software.exe!.text+0x3a407 (0x13eb407))
Argument 1/5
0038A828: 95 80 27 02 21 D5 2D 1A-0F D9 45 9F 75 EE 24 C4 "...'.!.-...E.u.$
↳ ."
Argument 2/5
0038A838: B1 27 7F 84 FE 41 37 86-C9 C0 00 CD CD CD CD CD ".'....A7
↳ ....."
Argument 3/5
0038BB40: CD CD CD CD CD CD CD CD-CD CD CD CD CD CD CD CD
↳ " . . . . . . . . . . ."
(0) software.exe!0x4339a0() -> 0x0
(0) software.exe!0x4339a0(0x38b920, 0x38a828, 0x38bb30, 0x10, 0x0) (called
↳ from software.exe!.text+0x33c0d (0x13e4c0d))
Argument 1/5
0038B920: CA 39 B1 85 75 1B 84 1F-F9 31 5E 39 72 13 EC 5D ".9..u....1^9r
↳ ..]"
Argument 2/5
0038A828: 95 80 27 02 21 D5 2D 1A-0F D9 45 9F 75 EE 24 C4 "...'.!.-...E.u.$
↳ ."
Argument 3/5
0038BB30: CD CD CD CD CD CD CD CD-CD CD CD CD CD CD CD CD
↳ " . . . . . . . . . . ."
(0) software.exe!0x4339a0() -> 0x0
Argument 3/5 difference
00000000: 45 00 20 00 4A 00 4F 00-48 00 4E 00 53 00 00 00 "E. .J.O.H.N.S
↳ ..."
(0) software.exe!0x4339a0(0x38b920, 0x0, 0x38b978, 0x10, 0x0) (called from
↳ software.exe!.text+0x33c0d (0x13e4c0d))
Argument 1/5
0038B920: 95 80 27 02 21 D5 2D 1A-0F D9 45 9F 75 EE 24 C4 "...'.!.-...E.u.$
↳ ."
Argument 3/5
0038B978: 95 80 27 02 21 D5 2D 1A-0F D9 45 9F 75 EE 24 C4 "...'.!.-...E.u.$
↳ ."
(0) software.exe!0x4339a0() -> 0x0
Argument 3/5 difference
00000000: B1 27 7F E4 9F 01 E3 81-CF C6 12 FB B9 7C F1 BC
↳ "...'. . . . . . . . . . |.. ."
PID=1984|Process software.exe exited. ExitCode=0 (0x0)

```

Тут мы можем увидеть входы в функцию *ProcessAndXorBlock()*, и выходы из нее.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



Это вывод из ф-ции во время первого вызова:

```
00000000: C7 39 4E 7B 33 1B D6 1F-B8 31 10 39 39 13 A5 5D ".9N↵
↳ {3....1.99..}"
```

Затем ф-ция *ProcessAndXorBlock()* вызывается с блоком нулевого размера, но с флагом 8 (*BT\_ReverseDirection*).

Второй вызов:

```
00000000: 45 00 20 00 4A 00 4F 00-48 00 4E 00 53 00 00 00 "E. .J.O.H.N.S↵
↳ ..."
```

Ох, тут есть знакомая нам строка!

Третий вызов:

```
00000000: B1 27 7F E4 9F 01 E3 81-CF C6 12 FB B9 7C F1 BC ↵
↳ ".!.....|.."
```

Первый вывод очень похож на первые 16 байт зашифрованного буфера.

Вывод первого вызова *ProcessAndXorBlock()*:

```
00000000: C7 39 4E 7B 33 1B D6 1F-B8 31 10 39 39 13 A5 5D ".9N↵
↳ {3....1.99..}"
```

Первые 16 байт зашифрованного буфера:

```
00000000: CA 39 B1 85 75 1B 84 1F F9 31 5E 39 72 13 EC 5D .9..u....1^9r..]
```

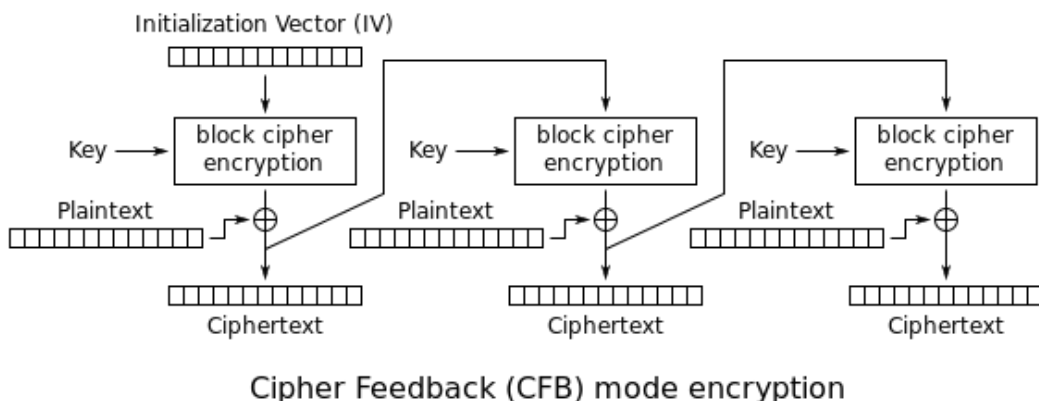
Тут слишком много одинаковых байт! Как так получается, что результат шифрования AES может быть очень похож на зашифрованный буфер в то время как это не шифрование, а скорее дешифрование?

### 8.7.5. Режим обратной связи по шифротексту

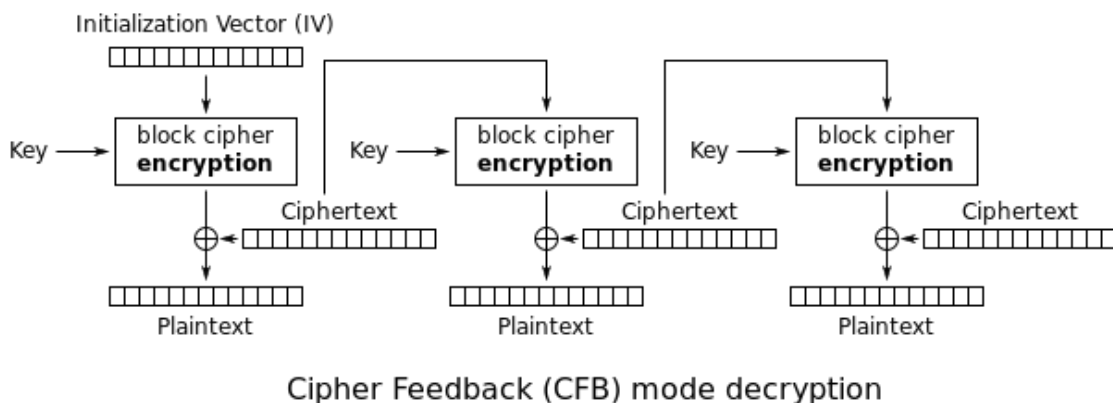
Ответ это **CFB!**<sup>21</sup>: в этом режиме, алгоритм AES используются не как алгоритм шифрования, а как устройство для генерации случайных данных с криптографической стойкостью. Само шифрование производится используя простую операцию XOR.

Вот алгоритм шифрования (иллюстрации взяты из Wikipedia):

<sup>21</sup>CFB!



И дешифрования:



Посмотрим: операция шифрования в AES генерирует 16 байт (или 128 бит) *случайных* данных, которые можно использовать во время применения операции XOR, но кто заставляет нас использовать все 16 байт? Если на последней итерации у нас 1 байт данных, давайте про-XOR-им 1 байт данных с 1 байтом сгенерированных *случайных* данных? Это приводит к важному свойству режима **CFB!**: данные не нужно выравнивать, данные произвольного размера могут быть зашифрованы и дешифрованы.

О, и вот почему все зашифрованные блоки не выровнены. И вот почему инструкция AESDEC никогда не вызывается.

Давайте попробуем дешифровать первый блок вручную, используя Питон. Режим **CFB!** также использует **IV!**<sup>22</sup>, как *seed* для **CSPRNG!**<sup>23</sup>. В нашем случае, **IV!** это блок, который шифруется на первой итерации:

<sup>22</sup> **IV!**

<sup>23</sup> **CSPRNG!**

```
0038B920: 01 00 00 00 FF FF FF FF-79 C1 69 0B 67 C1 04 7D ".....y.i.g↵
↳ ..}"
```

О, и нам нужно также восстановить ключ шифрования. В DLL есть AESKEYGENASSIST, и она вызывается, и используется в ф-ции `Rijndael::Base::UncheckedSetKey(): src`. Её легко найти в IDA и установить бряк-пойнт. Посмотрим:

```
... tracer.exe -l:filename.exe bpf=filename.exe!0x435c30,args:3,dump_args:0↵
↳ x10

Warning: no tracer.cfg file.
PID=2068|New process software.exe
no module registered with image base 0x77320000
no module registered with image base 0x76e20000
no module registered with image base 0x77320000
no module registered with image base 0x77220000
Warning: unknown (to us) INT3 breakpoint at ntdll.dll!↵
↳ LdrVerifyImageMatchesChecksum+0x96c (0x776c103b)
(0) software.exe!0x435c30(0x15e8000, 0x10, 0x14f808) (called from software.↵
↳ exe!.text+0x22fa1 (0x13d3fa1))
Argument 1/3
015E8000: CD C5 7E AD 28 5F 6D E1-CE 8F CC 29 B1 21 88 8E "...~.(~m....)↵
↳ !!"
Argument 3/3
0014F808: 38 82 58 01 C8 B9 46 00-01 D1 3C 01 00 F8 14 00 "8.X...F↵
↳ ...<....."
Argument 3/3 +0x0: software.exe!.rdata+0x5238
Argument 3/3 +0x8: software.exe!.text+0x1c101
(0) software.exe!0x435c30() -> 0x13c2801
PID=2068|Process software.exe exited. ExitCode=0 (0x0)
```

Так вот это ключ: `CD C5 7E AD 28 5F 6D E1-CE 8F CC 29 B1 21 88 8E`.

Во время ручного дешифрования мы получаем это:

```
00000000: 0D 00 FF FE 46 00 52 00 41 00 4E 00 4B 00 49 00 ....F.R.A.N.K.I↵
↳ .
00000010: 45 00 20 00 4A 00 4F 00 48 00 4E 00 53 00 66 66 E. .J.O.H.N.S.↵
↳ ff
00000020: 66 66 66 9E 61 40 D4 07 06 01 fff.a@....
```

Теперь это что-то читаемое! И теперь мы видим, почему было так много одинаковых байт во время первой итерации дешифрования: потому что в оригинальном тексте так много нулевых байт!

Дешифруем второй блок:

```
00000000: 17 98 D0 84 3A E9 72 4F DB 82 3F AD E9 3E 2A A8 .....r0↵
↳ ..?..>*.
00000010: 41 00 52 00 52 00 4F 00 4E 00 CD CC CC CC CC CC A.R.R.O.N↵
↳ .....
00000020: 1B 40 D4 07 06 01 .@....
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Третий, четвертый и пятый:

```
00000000: 5D 90 59 06 EF F4 96 B4 7C 33 A7 4A BE FF 66 AB ] .Y.....|3.J..f↵
    ↵ .
00000010: 49 00 47 00 47 00 53 00 00 00 00 00 00 C0 65 40 I.G.G.S.....↵
    ↵ e@
00000020: D4 07 06 01      ....
```

```
00000000: D3 15 34 5D 21 18 7C 6E AA F8 2D FE 38 F9 D7 4E ..4]!.|n..-.8..↵
    ↵ N
00000010: 41 00 20 00 44 00 4F 00 48 00 45 00 52 00 54 00 A. .D.O.H.E.R.T↵
    ↵ .
00000020: 59 00 48 E1 7A 14 AE FF 68 40 D4 07 06 02      Y.H.z...h@....
```

```
00000000: 1E 8B 90 0A 17 7B C5 52 31 6C 4E 2F DE 1B 27 19 .....{.R1lN↵
    ↵ ...'.
00000010: 41 00 52 00 43 00 55 00 53 00 00 00 00 00 60 A.R.C.U.S↵
    ↵ .....
00000020: 66 40 D4 07 06 03      f@....
```

Все блоки, похоже, дешифруются корректно, но не первые 16 байт.

### 8.7.6. Инициализирующий вектор

Что влияет на первые 16 байт?

Вернемся снова к алгоритму дешифрования **CFB!** ?? (стр. ??).

Мы видим что **IV!** может влиять на первую операцию дешифрования, но не на вторую, потому что во время второй итерации используется шифротекст от первой итерации, и в случае дешифрования, он такой же, не важно, какой был **IV!**

Так что, вероятно, **IV!** каждый раз разный. Используя мой tracer, я буду смотреть на первый вход во время дешифрования второго блока **XML!**-файла:

```
0038B920: 02 00 00 00 FE FF FF FF-79 C1 69 0B 67 C1 04 7D ".....y.i.g↵
    ↵ ..}"
```

...third:

```
0038B920: 03 00 00 00 FD FF FF FF-79 C1 69 0B 67 C1 04 7D ".....y.i.g↵
    ↵ ..}"
```

Похоже, первый и пятый байт каждый раз меняется. Я в итоге разобрался, что первое 32-битное число это просто OrderID из **XML!**-файла, и второе 32-битное число это тоже OrderID, но с отрицательным знаком. Остальные 8 байт не меняются. И вот я расшифровал всю БД: [https://beginners.re/paywall/RE4B-source/current-tree//examples/encrypted\\_DB1/decrypted.full.txt](https://beginners.re/paywall/RE4B-source/current-tree//examples/encrypted_DB1/decrypted.full.txt).

Питоновский скрипт, который я использовал: [https://beginners.re/paywall/RE4B-source/current-tree//examples/encrypted\\_DB1/decrypt\\_blocks.py](https://beginners.re/paywall/RE4B-source/current-tree//examples/encrypted_DB1/decrypt_blocks.py).

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Вероятно, автор хотел чтобы каждый блок шифровался немного иначе, так что он/она использовал OrderID как часть ключа. А еще можно было бы делать разный ключ для AES вместо **IV!**.

Так что теперь мы знаем, что **IV!** влияет только на первый блок во время дешифрования в режиме **CFB!**, это его особенность. Остальные блоки можно дешифровать не зная **IV!**, но используя ключ.

ОК, но почему режим **CFB!**? Очевидно, потому что самый первый пример на AES в CryptoPP wiki использует режим **CFB!**: [http://www.cryptopp.com/wiki/Advanced\\_Encryption\\_Standard#Encrypting\\_and\\_Decrypting\\_Using\\_AES](http://www.cryptopp.com/wiki/Advanced_Encryption_Standard#Encrypting_and_Decrypting_Using_AES). Вероятно, разработчик выбрал его из-за простоты: пример может шифровать/дешифровать текстовые строки произвольной длины, без выравнивания.

Очень похоже что автор этой программы просто скопипастил пример из страницы в CryptoPP wiki. Многие программисты так и делают.

Разница только в том что в примере в CryptoPP wiki **IV!** выбирается случайно, в то время как подобный индетерминизм не был допустимым для автора программы, которую мы сейчас разбираем, так что они решили инициализировать **IV!** используя ID заказа.

Теперь мы можем идти дальше, анализировать значение каждого байта дешифрованного блока.

### 8.7.7. Структура буфера

Возьмем первые 4 байта дешифрованных блоков:

```

00000000: 0D 00 FF FE 46 00 52 00 41 00 4E 00 4B 00 49 00  ....F.R.A.N.K.I↵
    ↵ .
00000010: 45 00 20 00 4A 00 4F 00 48 00 4E 00 53 00 66 66  E. .J.O.H.N.S.↵
    ↵ ff
00000020: 66 66 66 9E 61 40 D4 07 06 01                      fff.a@....

00000000: 0B 00 FF FE 4C 00 4F 00 52 00 49 00 20 00 42 00  ....L.O.R.I. .B↵
    ↵ .
00000010: 41 00 52 00 52 00 4F 00 4E 00 CD CC CC CC CC CC  A.R.R.O.N↵
    ↵ .....
00000020: 1B 40 D4 07 06 01                      .@....

00000000: 0A 00 FF FE 47 00 41 00 52 00 59 00 20 00 42 00  ....G.A.R.Y. .B↵
    ↵ .
00000010: 49 00 47 00 47 00 53 00 00 00 00 00 00 C0 65 40  I.G.G.S.....↵
    ↵ e@
00000020: D4 07 06 01                      ....

00000000: 0F 00 FF FE 4D 00 45 00 4C 00 49 00 4E 00 44 00  ....M.E.L.I.N.D↵
    ↵ .
00000010: 41 00 20 00 44 00 4F 00 48 00 45 00 52 00 54 00  A. .D.O.H.E.R.T↵
    ↵ .
00000020: 59 00 48 E1 7A 14 AE FF 68 40 D4 07 06 02          Y.H.z...h@....

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Легко увидеть строки закодированные в UTF-16, это имена и фамилии. Первый байт (или 16-битное слово) похоже это просто длина строки, мы можем проверить это визуально. *FF FE* это, похоже, **BOM!** в Уникоде.

После каждой строки есть еще 12 байт.

Используя этот скрипт ([https://beginners.re/paywall/RE4B-source/current-tree/examples/encrypted\\_DB1/dump\\_buffer\\_rest.py](https://beginners.re/paywall/RE4B-source/current-tree/examples/encrypted_DB1/dump_buffer_rest.py)) я получил случайную выборку из *ХВОСТОВ*:

```
dennis@...$ python decrypt.py encrypted.xml | shuf | head -20
00000000: 48 E1 7A 14 AE 5F 62 40 DD 07 05 08 H.z..._b@....
00000000: 00 00 00 00 00 40 5A 40 DC 07 08 18 .....@Z@....
00000000: 00 00 00 00 00 80 56 40 D7 07 0B 04 .....V@....
00000000: 00 00 00 00 00 60 61 40 D7 07 0C 1C .....a@....
00000000: 00 00 00 00 00 20 63 40 D9 07 05 18 .....c@....
00000000: 3D 0A D7 A3 70 FD 34 40 D7 07 07 11 =...p.4@....
00000000: 00 00 00 00 00 A0 63 40 D5 07 05 19 .....c@....
00000000: CD CC CC CC CC 3C 5C 40 D7 07 08 11 .....@....
00000000: 66 66 66 66 66 FE 62 40 D4 07 06 05 fffff.b@....
00000000: 1F 85 EB 51 B8 FE 40 40 D6 07 09 1E ...Q...@....
00000000: 00 00 00 00 00 40 5F 40 DC 07 02 18 .....@_@....
00000000: 48 E1 7A 14 AE 9F 67 40 D8 07 05 12 H.z...g@....
00000000: CD CC CC CC CC 3C 5E 40 DC 07 01 07 .....^@....
00000000: 00 00 00 00 00 00 67 40 D4 07 0B 0E .....g@....
00000000: 00 00 00 00 00 40 51 40 DC 07 04 0B .....@Q@....
00000000: 00 00 00 00 00 40 56 40 D7 07 07 0A .....@V@....
00000000: 8F C2 F5 28 5C 7F 55 40 DB 07 01 16 ...(.U@....
00000000: 00 00 00 00 00 00 32 40 DB 07 06 09 .....2@....
00000000: 66 66 66 66 66 7E 66 40 D9 07 0A 06 fffff~f@....
00000000: 48 E1 7A 14 AE DF 68 40 D5 07 07 16 H.z...h@....
```

Видим что байты 0x40 и 0x07 присутствуют в каждом хвосте. Самый последний байт всегда в пределах 1..0x1F (1..31), я проверил. Предпоследний байт всегда в пределах 1..0xC (1..12). Ух, это выглядит как дата! Год может быть представлен как 16-битное значение, и может быть последние 4 байта это дата (16 бит для года, 8 бит для месяца и еще 8 для дня)? 0x7DD это 2013, 0x7D5 это 2005, и т. д. Похоже, нормально. Это дата. Там есть еще 8 байт. Судя по тому факту что БД называется *orders* (заказы), может быть здесь присутствует сумма? Я сделал попытку интерпретировать их как числа с плавающей точкой двойной точности в формате IEEE 754, и вывести все значения!

Некоторые:

```
71.0
134.0
51.95
53.0
121.99
96.95
98.95
15.95
85.95
184.99
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

94.95
29.95
85.0
36.0
130.99
115.95
87.99
127.95
114.0
150.95

```

Похоже на правду!

Теперь мы можем вывести имена, суммы и даты.

```

plain:
00000000: 0D 00 FF FE 46 00 52 00 41 00 4E 00 4B 00 49 00  ....F.R.A.N.K.I.↵
  ↵ .
00000010: 45 00 20 00 4A 00 4F 00 48 00 4E 00 53 00 66 66  E. .J.O.H.N.S.↵
  ↵ ff
00000020: 66 66 66 9E 61 40 D4 07 06 01                      fff.a@....
OrderID= 1 name= FRANKIE JOHNS sum= 140.95 date= 2004 / 6 / 1

plain:
00000000: 0B 00 FF FE 4C 00 4F 00 52 00 49 00 20 00 42 00  ....L.O.R.I. .B↵
  ↵ .
00000010: 41 00 52 00 52 00 4F 00 4E 00 CD CC CC CC CC CC  A.R.R.O.N↵
  ↵ .....
00000020: 1B 40 D4 07 06 01                      .@....
OrderID= 2 name= LORI BARRON sum= 6.95 date= 2004 / 6 / 1

plain:
00000000: 0A 00 FF FE 47 00 41 00 52 00 59 00 20 00 42 00  ....G.A.R.Y. .B↵
  ↵ .
00000010: 49 00 47 00 47 00 53 00 00 00 00 00 00 C0 65 40  I.G.G.S.....↵
  ↵ e@
00000020: D4 07 06 01                      ....
OrderID= 3 name= GARY BIGGS sum= 174.0 date= 2004 / 6 / 1

plain:
00000000: 0F 00 FF FE 4D 00 45 00 4C 00 49 00 4E 00 44 00  ....M.E.L.I.N.D↵
  ↵ .
00000010: 41 00 20 00 44 00 4F 00 48 00 45 00 52 00 54 00  A. .D.O.H.E.R.T↵
  ↵ .
00000020: 59 00 48 E1 7A 14 AE FF 68 40 D4 07 06 02          Y.H.z...h@....
OrderID= 4 name= MELINDA DOHERTY sum= 199.99 date= 2004 / 6 / 2

plain:
00000000: 0B 00 FF FE 4C 00 45 00 4E 00 41 00 20 00 4D 00  ....L.E.N.A. .M↵
  ↵ .
00000010: 41 00 52 00 43 00 55 00 53 00 00 00 00 00 00 60  A.R.C.U.S↵
  ↵ .....
00000020: 66 40 D4 07 06 03                      f@....
OrderID= 5 name= LENA MARCUS sum= 179.0 date= 2004 / 6 / 3

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

См. еще: [https://beginners.re/paywall/RE4B-source/current-tree//examples/encrypted\\_DB1/decrypted.full.with.data.txt](https://beginners.re/paywall/RE4B-source/current-tree//examples/encrypted_DB1/decrypted.full.with.data.txt). Или отфильтрованные: [https://beginners.re/paywall/RE4B-source/current-tree//examples/encrypted\\_DB1/decrypted.short.txt](https://beginners.re/paywall/RE4B-source/current-tree//examples/encrypted_DB1/decrypted.short.txt). Похоже всё корректно.

Это что-то вроде сериализации в **ООР!**, т.е., упаковка значений с разными типами в бинарный буфер для хранения и/или передачи.

### 8.7.8. Шум в конце

Остался только один вопрос, иногда *хвост* длиннее:

```
00000000: 0E 00 FF FE 54 00 48 00 45 00 52 00 45 00 53 00  ....T.H.E.R.E.S↵
    ↵ .
00000010: 45 00 20 00 54 00 55 00 54 00 54 00 4C 00 45 00  E. .T.U.T.T.L.E↵
    ↵ .
00000020: 66 66 66 66 66 1E 63 40 D4 07 07 1A 00 07 07 19  fffff.c@↵
    ↵ .....
OrderID= 172 name= THERESE TUTTLE sum= 152.95 date= 2004 / 7 / 26
```

(Байты *00 07 07 19* не используются и являются балластом.)

```
00000000: 0C 00 FF FE 4D 00 45 00 4C 00 41 00 4E 00 49 00  ....M.E.L.A.N.I↵
    ↵ .
00000010: 45 00 20 00 4B 00 49 00 52 00 4B 00 00 00 00 00  E. .K.I.R.K↵
    ↵ .....
00000020: 00 20 64 40 D4 07 09 02 00 02                      . d@.....
OrderID= 286 name= MELANIE KIRK sum= 161.0 date= 2004 / 9 / 2
```

(*00 02* не используются.)

После близкого рассмотрения мы можем видеть, что шум в конце *хвоста* просто остался от предыдущего шифрования!

Вот два идущих подряд буфера:

```
00000000: 10 00 FF FE 42 00 4F 00 4E 00 4E 00 49 00 45 00  ....B.O.N.N.I.E↵
    ↵ .
00000010: 20 00 47 00 4F 00 4C 00 44 00 53 00 54 00 45 00  .G.O.L.D.S.T.E↵
    ↵ .
00000020: 49 00 4E 00 9A 99 99 99 99 79 46 40 D4 07 07 19  I.N.....yF@↵
    ↵ ....
OrderID= 171 name= BONNIE GOLDSTEIN sum= 44.95 date= 2004 / 7 / 25

00000000: 0E 00 FF FE 54 00 48 00 45 00 52 00 45 00 53 00  ....T.H.E.R.E.S↵
    ↵ .
00000010: 45 00 20 00 54 00 55 00 54 00 54 00 4C 00 45 00  E. .T.U.T.T.L.E↵
    ↵ .
00000020: 66 66 66 66 66 1E 63 40 D4 07 07 1A 00 07 07 19  fffff.c@↵
    ↵ .....
OrderID= 172 name= THERESE TUTTLE sum= 152.95 date= 2004 / 7 / 26
```

(Последние байты *07 07 19* скопированы из предыдущего незашифрованного буфера.)

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



Еще два подряд идущих буфера:

```

00000000: 0D 00 FF FE 4C 00 4F 00 52 00 45 00 4E 00 45 00  ....L.O.R.E.N.E↵
    ↵ .
00000010: 20 00 4F 00 54 00 4F 00 4F 00 4C 00 45 00 CD CC  .O.T.O.O.L.E↵
    ↵ ...
00000020: CC CC CC 3C 5E 40 D4 07 09 02  ...<^@....
OrderID= 285 name= LORENE OTTOLE sum= 120.95 date= 2004 / 9 / 2

00000000: 0C 00 FF FE 4D 00 45 00 4C 00 41 00 4E 00 49 00  ....M.E.L.A.N.I↵
    ↵ .
00000010: 45 00 20 00 4B 00 49 00 52 00 4B 00 00 00 00 00  E. .K.I.R.K↵
    ↵ .....
00000020: 00 20 64 40 D4 07 09 02 00 02  . d@.....
OrderID= 286 name= MELANIE KIRK sum= 161.0 date= 2004 / 9 / 2

```

Последний байт 02 был скопирован из предыдущего незашифрованного буфера.

Возможно, что буфер использующийся для шифрования глобальный и/или не очищается перед каждым шифрованием. Размер последнего буфера тоже как-то хаотично меняется, тем не менее, ошибка не была отловлена потому что она не влияет на процесс дешифрования, который просто игнорирует шум в конце. Эта распространенная ошибка. Она была даже в OpenSSL (ошибка Heartbleed).

### 8.7.9. Вывод

Итог: каждый практикующий реверс-инженер должен быть знаком с основными алгоритмами шифрования, а также с основными режимами шифрования. Некоторые книги об этом: ?? (стр. ??).

*Зашифрованное* содержимое БД было искусственно мною создано ради демонстрации. Я использовал наиболее популярные имена и фамилии в США, отсюда: <http://stackoverflow.com/questions/1803628/raw-list-of-person-names>, и скомбинировал их случайным образом. Даты и суммы были сгенерированы случайным образом.

Все файлы использованные в этой части здесь: [https://beginners.re/paywall/RE4B-source/current-tree//examples/encrypted\\_DB1](https://beginners.re/paywall/RE4B-source/current-tree//examples/encrypted_DB1).

Тем не менее, многие особенности как здесь, я наблюдал в настоящем ПО. Этот пример основан на них.

### 8.7.10. Post Scriptum: перебор всех IV!

Пример, который вы только что видели, был искусственно создан, но основан на настоящем ПО которое я разбирал. Когда я над ним работал, я в начале заметил, что **IV!** генерируется используя некоторые 32-битное число, и я не мог найти связь между этим числом и OrderID. Так что я приготовился использовать полный перебор, который тут действительно возможен.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Это не проблема перебрать все 32-битные значения и попробовать каждое как основу для **IV!**. Затем вы дешифруете первый 16-байтный блок и проверяете нулевые байты, которые всегда находятся на одних и тех же местах.

## 8.8. Разгон майнера биткоинов Cointerra

Был такой майнер биткоинов Cointerra, выглядящий так:

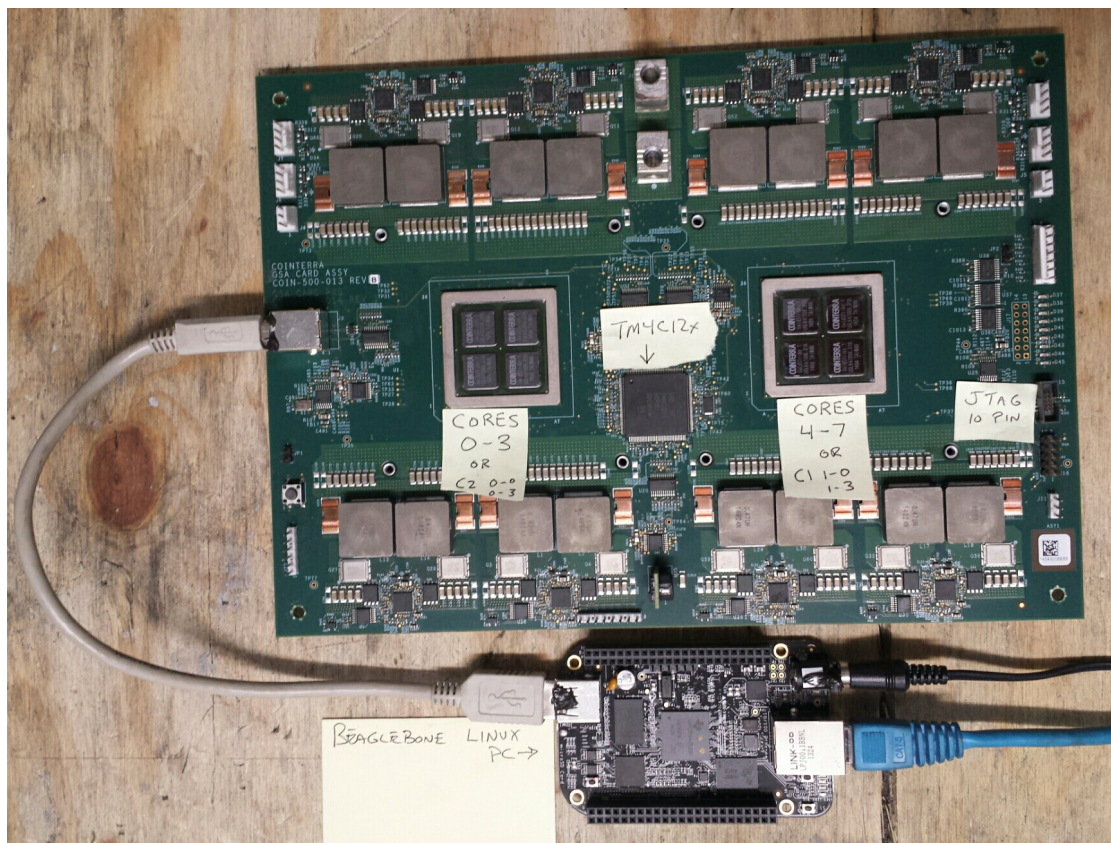


Рис. 8.14: Board

И была также (возможно утекшая) утилита<sup>24</sup> которая могла выставить тактовую частоту платы. Она запускается на дополнительной плате BeagleBone на ARM с Linux (маленькая плата внизу фотографии).

И у автора (этих строк) однажды спросили, можно ли хакнуть эту утилиту и посмотреть, какие частоты можно выставить, и какие нет. И можно ли твикнуть её?

<sup>24</sup>Можно скачать здесь: [https://beginners.re/paywall/RE4B-source/current-tree//examples/bitcoin\\_miner/files/cointool-overclock](https://beginners.re/paywall/RE4B-source/current-tree//examples/bitcoin_miner/files/cointool-overclock)

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Утилиту нужно запускать так: `./cointool-overclock 0 0 900`, где 900 это частота в МГц. Если частота слишком большая, утилита выведет ошибку «Error with arguments» и закончит работу.

Вот фрагмент кода вокруг ссылки на текстовую строку «Error with arguments»:

```

...
.text:0000ABC4      STR      R3, [R11,#var_28]
.text:0000ABC8      MOV      R3, #optind
.text:0000ABD0      LDR      R3, [R3]
.text:0000ABD4      ADD      R3, R3, #1
.text:0000ABD8      MOV      R3, R3,LSL#2
.text:0000ABDC      LDR      R2, [R11,#argv]
.text:0000ABE0      ADD      R3, R2, R3
.text:0000ABE4      LDR      R3, [R3]
.text:0000ABE8      MOV      R0, R3 ; nptr
.text:0000ABEC      MOV      R1, #0 ; endptr
.text:0000ABF0      MOV      R2, #0 ; base
.text:0000ABF4      BL       strtoll
.text:0000ABF8      MOV      R2, R0
.text:0000ABFC      MOV      R3, R1
.text:0000AC00      MOV      R3, R2
.text:0000AC04      STR      R3, [R11,#var_2C]
.text:0000AC08      MOV      R3, #optind
.text:0000AC10      LDR      R3, [R3]
.text:0000AC14      ADD      R3, R3, #2
.text:0000AC18      MOV      R3, R3,LSL#2
.text:0000AC1C      LDR      R2, [R11,#argv]
.text:0000AC20      ADD      R3, R2, R3
.text:0000AC24      LDR      R3, [R3]
.text:0000AC28      MOV      R0, R3 ; nptr
.text:0000AC2C      MOV      R1, #0 ; endptr
.text:0000AC30      MOV      R2, #0 ; base
.text:0000AC34      BL       strtoll
.text:0000AC38      MOV      R2, R0
.text:0000AC3C      MOV      R3, R1
.text:0000AC40      MOV      R3, R2
.text:0000AC44      STR      R3, [R11,#third_argument]
.text:0000AC48      LDR      R3, [R11,#var_28]
.text:0000AC4C      CMP      R3, #0
.text:0000AC50      BLT     errors_with_arguments
.text:0000AC54      LDR      R3, [R11,#var_28]
.text:0000AC58      CMP      R3, #1
.text:0000AC5C      BGT     errors_with_arguments
.text:0000AC60      LDR      R3, [R11,#var_2C]
.text:0000AC64      CMP      R3, #0
.text:0000AC68      BLT     errors_with_arguments
.text:0000AC6C      LDR      R3, [R11,#var_2C]
.text:0000AC70      CMP      R3, #3
.text:0000AC74      BGT     errors_with_arguments
.text:0000AC78      LDR      R3, [R11,#third_argument]
.text:0000AC7C      CMP      R3, #0x31

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:0000AC80 BLE errors_with_arguments
.text:0000AC84 LDR R2, [R11,#third_argument]
.text:0000AC88 MOV R3, #950
.text:0000AC8C CMP R2, R3
.text:0000AC90 BGT errors_with_arguments
.text:0000AC94 LDR R2, [R11,#third_argument]
.text:0000AC98 MOV R3, #0x51EB851F
.text:0000ACA0 SMULL R1, R3, R3, R2
.text:0000ACA4 MOV R1, R3,ASR#4
.text:0000ACA8 MOV R3, R2,ASR#31
.text:0000ACAC RSB R3, R3, R1
.text:0000ACB0 MOV R1, #50
.text:0000ACB4 MUL R3, R1, R3
.text:0000ACB8 RSB R3, R3, R2
.text:0000ACBC CMP R3, #0
.text:0000ACC0 BEQ loc_ACEC
.text:0000ACC4
.text:0000ACC4 errors_with_arguments
.text:0000ACC4
.text:0000ACC4 LDR R3, [R11,#argv]
.text:0000ACC8 LDR R3, [R3]
.text:0000ACCC MOV R0, R3 ; path
.text:0000ACD0 BL __xpg_basename
.text:0000ACD4 MOV R3, R0
.text:0000ACD8 MOV R0, #aSErrorWithArgu ; format
.text:0000ACE0 MOV R1, R3
.text:0000ACE4 BL printf
.text:0000ACE8 B loc_ADD4
.text:0000ACEC ; -----
.text:0000ACEC
.text:0000ACEC loc_ACEC ; CODE XREF: main+66C
.text:0000ACEC LDR R2, [R11,#third_argument]
.text:0000ACF0 MOV R3, #499
.text:0000ACF4 CMP R2, R3
.text:0000ACF8 BGT loc_AD08
.text:0000ACFC MOV R3, #0x64
.text:0000AD00 STR R3, [R11,#unk_constant]
.text:0000AD04 B jump_to_write_power
.text:0000AD08 ; -----
.text:0000AD08
.text:0000AD08 loc_AD08 ; CODE XREF: main+6A4
.text:0000AD08 LDR R2, [R11,#third_argument]
.text:0000AD0C MOV R3, #799
.text:0000AD10 CMP R2, R3
.text:0000AD14 BGT loc_AD24
.text:0000AD18 MOV R3, #0x5F
.text:0000AD1C STR R3, [R11,#unk_constant]
.text:0000AD20 B jump_to_write_power
.text:0000AD24 ; -----
.text:0000AD24
.text:0000AD24 loc_AD24 ; CODE XREF: main+6C0
.text:0000AD24 LDR R2, [R11,#third_argument]
.text:0000AD28 MOV R3, #899

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:0000AD2C      CMP     R2, R3
.text:0000AD30      BGT     loc_AD40
.text:0000AD34      MOV     R3, #0x5A
.text:0000AD38      STR     R3, [R11,#unk_constant]
.text:0000AD3C      B       jump_to_write_power
.text:0000AD40      ; -----
.text:0000AD40      loc_AD40      ; CODE XREF: main+6DC
.text:0000AD40      LDR     R2, [R11,#third_argument]
.text:0000AD44      MOV     R3, #999
.text:0000AD48      CMP     R2, R3
.text:0000AD4C      BGT     loc_AD5C
.text:0000AD50      MOV     R3, #0x55
.text:0000AD54      STR     R3, [R11,#unk_constant]
.text:0000AD58      B       jump_to_write_power
.text:0000AD5C      ; -----
.text:0000AD5C      loc_AD5C      ; CODE XREF: main+6F8
.text:0000AD5C      LDR     R2, [R11,#third_argument]
.text:0000AD60      MOV     R3, #1099
.text:0000AD64      CMP     R2, R3
.text:0000AD68      BGT     jump_to_write_power
.text:0000AD6C      MOV     R3, #0x50
.text:0000AD70      STR     R3, [R11,#unk_constant]
.text:0000AD74      jump_to_write_power      ; CODE XREF: main+6B0
.text:0000AD74      ; main+6CC ...
.text:0000AD74      LDR     R3, [R11,#var_28]
.text:0000AD78      UXTB   R1, R3
.text:0000AD7C      LDR     R3, [R11,#var_2C]
.text:0000AD80      UXTB   R2, R3
.text:0000AD84      LDR     R3, [R11,#unk_constant]
.text:0000AD88      UXTB   R3, R3
.text:0000AD8C      LDR     R0, [R11,#third_argument]
.text:0000AD90      UXTH   R0, R0
.text:0000AD94      STR     R0, [SP,#0x44+var_44]
.text:0000AD98      LDR     R0, [R11,#var_24]
.text:0000AD9C      BL     write_power
.text:0000ADA0      LDR     R0, [R11,#var_24]
.text:0000ADA4      MOV     R1, #0x5A
.text:0000ADA8      BL     read_loop
.text:0000ADAC      B       loc_ADD4
...

.rodata:0000B378 aSErrorWithArgu DCB "%s: Error with arguments",0xA,0 ; DATA
XREF: main+684
...

```

Имена ф-ций присутствовали в отладочной информации в оригинальном исполняемом файле, такие как `write_power`, `read_loop`. Но имена меткам внутри ф-ции дал я.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



Имя `optind` звучит знакомо. Это библиотека `getopt` из \*NIX предназначенная для парсинга командной строки — и это то, что внутри и происходит. Затем, третий аргумент (где передается значение частоты) конвертируется из строку в число используя вызов ф-ции `strtoll()`.

Значение затем сравнивается с разными константами. На `0xACEC` есть проверка, меньше ли оно или равно 499, и если это так, то `0x64` будет передано в ф-цию `write_power()` (которая посылает команду через USB используя `send_msg()`). Если значение больше 499, происходит переход на `0xAD08`.

На `0xAD08` есть проверка, меньше ли оно или равно 799. Если это так, то `0x5F` передается в ф-цию `write_power()`.

Есть еще проверки: на 899 на `0xAD24`, на 999 на `0xAD40`, и наконец, на 1099 на `0xAD5C`. Если входная частота меньше или равна 1099, `0x50` (на `0xAD6C`) будет передано в ф-цию `write_power()`. И тут что-то вроде баги. Если значение все еще больше 1099, само значение будет передано в ф-цию `write_power()`. Но с другой стороны это не бага, потому что мы не можем попасть сюда: значение в начале проверяется с 950 на `0xAC88`, и если оно больше, выводится сообщение об ошибке и утилита заканчивает работу.

Вот таблица между частотами в МГц и значениями передаваемыми в ф-цию `write_power()`:

МГц	шестнадцатеричное представление	десятичное
499MHz	0x64	100
799MHz	0x5f	95
899MHz	0x5a	90
999MHz	0x55	85
1099MHz	0x50	80

Как видно, значение передаваемое в плату постепенно уменьшается с ростом частоты.

Видно что значение в 950МГц это жесткий предел, по крайней мере в этой утилите. Можно ли её обмануть?

Вернемся к этому фрагменту кода:

```
.text:0000AC84    LDR    R2, [R11,#third_argument]
.text:0000AC88    MOV    R3, #950
.text:0000AC8C    CMP    R2, R3
.text:0000AC90    BGT    errors_with_arguments ; Я пропатчил здесь на 00
                00 00 00
```

Нам нужно как-то запретить инструкцию перехода `BGT` на `0xAC90`. И это ARM в режиме ARM, потому что, как мы видим, все адреса увеличиваются на 4, т.е., длина каждой инструкции это 4 байта. Инструкция `NOP` (нет операции) в режиме ARM это просто 4 нулевых байта: `00 00 00 00`. Так что, записывая 4 нуля по адресу `0xAC90` (или по физическому смещению в файле: `0x2C90`) мы можем выключить эту проверку.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Теперь можно выставлять частоты вплоть до 1050МГц. И даже больше, но из-за ошибки, если входное значение больше 1099, значение в МГц, как есть, будет передано в плату, что неправильно.

Дальше я не разобрался, но если бы продолжил, я бы уменьшал значение передаваемое в ф-цию `write_power()`.

Теперь страшный фрагмент кода, который я в начале пропустил:

```
.text:0000AC94      LDR      R2, [R11,#third_argument]
.text:0000AC98      MOV      R3, #0x51EB851F
.text:0000ACA0      SMULL   R1, R3, R3, R2 ; R3=3rg_arg/3.125
.text:0000ACA4      MOV      R1, R3,ASR#4 ; R1=R3/16=3rg_arg/50
.text:0000ACA8      MOV      R3, R2,ASR#31 ; R3=MSB(3rg_arg)
.text:0000ACAC      RSB     R3, R3, R1 ; R3=3rd_arg/50
.text:0000ACB0      MOV      R1, #50
.text:0000ACB4      MUL     R3, R1, R3 ; R3=50*(3rd_arg/50)
.text:0000ACB8      RSB     R3, R3, R2
.text:0000ACBC      CMP     R3, #0
.text:0000ACC0      BEQ     loc_ACEC
.text:0000ACC4
.text:0000ACC4 errors_with_arguments
```

Здесь используется деление через умножение, и константа 0x51EB851F. Я написал для себя простой программистский калькулятор<sup>25</sup> И там есть возможность вычислять обратное число по модулю.

```
modinv32(0x51EB851F)
Warning, result is not integer: 3.125000
(unsigned) dec: 3 hex: 0x3 bin: 11
```

Это значит что инструкция SMULL на 0xACA0 просто делит 3-й аргумент на 3.125. На самом деле, все что делает ф-ция `modinv32()` в моем калькуляторе, это:

$$\frac{1}{\frac{input}{2^{32}}} = \frac{2^{32}}{input}$$

Потом там есть дополнительные сдвиги и теперь мы видим что 3-й аргумент просто делится на 50. И затем умножается снова на 50. Зачем? Это простейшая проверка, можно ли делить входное значение на 50 без остатка. Если значение этого выражения ненулевое,  $x$  не может быть разделено на 50 без остатка:

$$x - \left(\left(\frac{x}{50}\right) \cdot 50\right)$$

На самом деле, это простой способ вычисления остатка от деления.

И затем, если остаток ненулевой, выводится сообщение об ошибке. Так что эта утилита берет значения частот вроде 850, 900, 950, 1000, и т. д., но не 855 или 911.

<sup>25</sup><https://yurichev.com/progcalc/>

Вот и всё! Если вы делаете что-то такое, имейте ввиду, что это может испортить вашу плату, как и в случае разгона чипов вроде **CPU!**, **GPU!**<sup>26</sup>, и т. д. Если у вас есть плата Cointerra, делайте всё это на свой собственный риск!

## 8.9. Взлом простого шифровальщика исполняемого кода

У нас есть исполняемый файл, зашифрованный относительно простым шифрованием. [Вот он](#) (здесь осталась только исполняемая секция).

Прежде всего, всё что делает ф-ция шифрования это прибавляет номер позиции в буфере к байту. Вот как это можно описать:

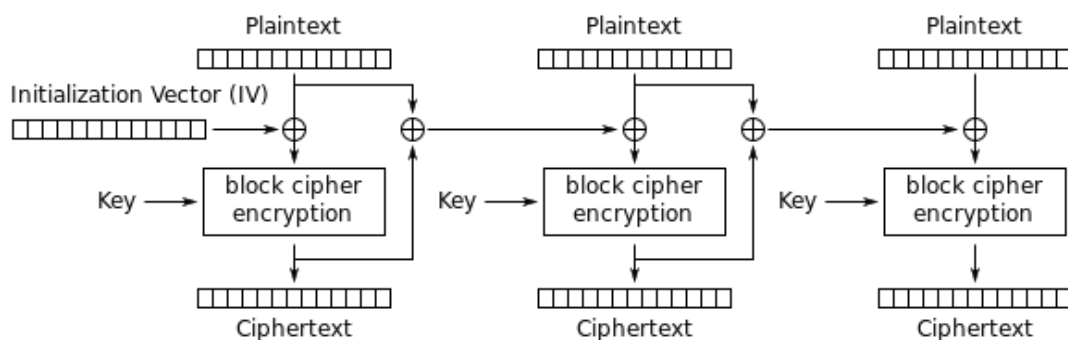
Листинг 8.9: Скрипт на Питоне

```
#!/usr/bin/env python
def e(i, k):
    return chr ((ord(i)+k) % 256)

def encrypt(buf):
    return e(buf[0], 0)+ e(buf[1], 1)+ e(buf[2], 2) + e(buf[3], 3)+ e(buf[4], 4)+ e(buf[5], 5)+ e(buf[6], 6)+ e(buf[7], 7)+
    e(buf[8], 8)+ e(buf[9], 9)+ e(buf[10], 10)+ e(buf[11], 11)+ e(buf[12], 12)+ e(buf[13], 13)+ e(buf[14], 14)+ e(buf[15], 15)
```

Так, если вы зашифровываете буфер с 16-ю нулями, вы получаете 0, 1, 2, 3 ... 12, 13, 14, 15.

Также здесь используется “Propagating Cipher Block Chaining” (PCBC) (режим распространяющегося сцепления блоков шифра), и вот как он работает:



Propagating Cipher Block Chaining (PCBC) mode encryption

Рис. 8.15: Шифрование в режиме распространяющегося сцепления блоков шифра (иллюстрация взята из Wikipedia)

<sup>26</sup>GPU!



Проблема восстановить **IV!**. Полный перебор не подходит, потому что **IV!** слишком длинный (16 байт). Посмотрим, сможем ли мы восстановить **IV!** для произвольного зашифрованного исполняемого файла?

Давайте попробуем простой частотный анализ. Это 32-битный исполняемый x86 код, так что попробуем собрать статистику о наиболее частых байтах и опкодах. Я пробовал огромный файл oracle.exe из Oracle RDBMS версии 11.2 для windows x86 и нашел, что самый часто встречающийся байт (и это не удивляет) это ноль ( 10%). Другой часто встречающийся байт (снова, не удивительно) 0xFF ( 5%). Следующий 0x8B ( 5%).

0x8B это опкод инструкции MOV, и это действительно одна из самых используемых инструкций в x86. А что насчет популярности нулевого байта? Если компилятору нужно закодировать значение более 127, он будет использовать 32-битные значения (displacement) вместо 8-битных, но большие значения редки (?? (стр. ??)), так что они дополняются нулями. Это справедливо как минимум для LEA, MOV, PUSH, CALL.

Например:

8D B0 28 01 00 00	lea	esi, [eax+128h]
8D BF 40 38 00 00	lea	edi, [edi+3840h]

Смещения (displacements) более 127 очень популярны, но они редко превышают 0x10000 (действительно, настолько большие буфера/структуры также редки).

Та же история с MOV, большие константы редки, наиболее часто используемые это 0, 1, 10, 100,  $2^n$ , и т. д. Компилятору приходится дополнять маленькие константы нулями, чтобы представить их в виде 32-битных значений.

BF 02 00 00 00	mov	edi, 2
BF 01 00 00 00	mov	edi, 1

Теперь о 00 и FF вместе: переходы (включая условные) и вызовы могут переходить по адресам вперед или назад, но очень часто, в пределах текущего исполняемого модуля. Если вперед, то смещение (displacement) не очень большое и дополняется нулями. Если назад, то смещение представляется отрицательным значением, так что дополняется байтами 0xFF. Например, перейти вперед:

E8 43 0C 00 00	call	_function1
E8 5C 00 00 00	call	_function2
0F 84 F0 0A 00 00	jz	loc_4F09A0
0F 84 EB 00 00 00	jz	loc_4EFBB8

Назад:

E8 79 0C FE FF	call	_function1
E8 F4 16 FF FF	call	_function2
0F 84 F8 FB FF FF	jz	loc_8212BC
0F 84 06 FD FF FF	jz	loc_FF1E7D

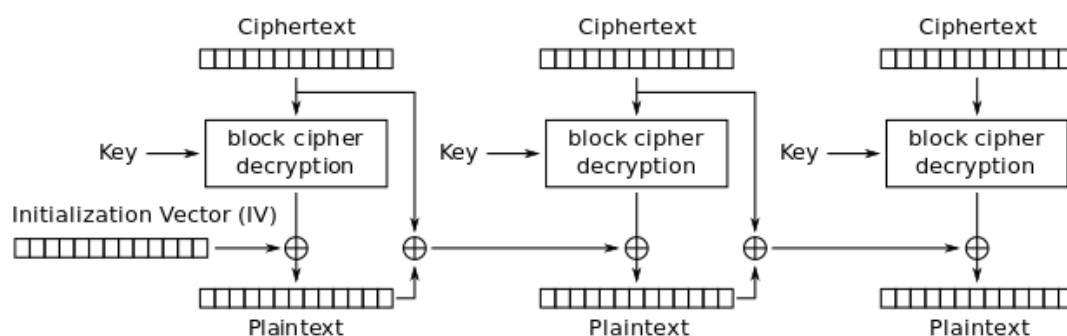
---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Байт 0xFF часто встречается в отрицательных смещениях вроде этих:

8D 85 1E FF FF FF	lea	eax,	[ebp-0E2h]
8D 95 F8 5C FF FF	lea	edx,	[ebp-0A308h]

Пока всё понятно. Нам теперь нужно пробовать разные 16-байтные ключи, дешифровать исполняемую секцию и измерять, как часто встречаются байты 0, 0xFF и 0x8B. Также будем держать на виду схему, как работает дешифрование в режиме PCBC:



Propagating Cipher Block Chaining (PCBC) mode decryption

Рис. 8.16: Дешифрование в режиме распространяющегося сцепления блоков шифра (иллюстрация взята из Wikipedia)

Хорошие новости в том, что нам не нужно дешифровать весь кусок данных, но только ломтик за ломтиком, точно так же, как я это делал в моем предыдущем примере: ?? (стр. ??).

Теперь я пробую все возможные байты (0..255) для каждого байта в ключе и просто выбираю тот байт, от которого будет максимальное число байтов 0x0/0xFF/0x8B в дешифрованном ломтике:

```
#!/usr/bin/env python
import sys, hexdump, array, string, operator

KEY_LEN=16

def chunks(l, n):
    # split n by l-byte chunks
    # https://stackoverflow.com/q/312443
    n = max(1, n)
    return [l[i:i + n] for i in range(0, len(l), n)]

def read_file(fname):
    file=open(fname, mode='rb')
    content=file.read()
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

file.close()
return content

def decrypt_byte (c, key):
    return chr((ord(c)-key) % 256)

def XOR_PCBC_step (IV, buf, k):
    prev=IV
    rt=""
    for c in buf:
        new_c=decrypt_byte(c, k)
        plain=chr(ord(new_c)^ord(prev))
        prev=chr(ord(c)^ord(plain))
        rt=rt+plain
    return rt

each_Nth_byte=[""]*KEY_LEN

content=read_file(sys.argv[1])
# split input by 16-byte chunks:
all_chunks=chunks(content, KEY_LEN)
for c in all_chunks:
    for i in range(KEY_LEN):
        each_Nth_byte[i]=each_Nth_byte[i] + c[i]

# try each byte of key
for N in range(KEY_LEN):
    print "N=", N
    stat={}
    for i in range(256):
        tmp_key=chr(i)
        tmp=XOR_PCBC_step(tmp_key,each_Nth_byte[N], N)
        # count 0, FFs and 8Bs in decrypted buffer:
        important_bytes=tmp.count('\x00')+tmp.count('\xFF')+tmp.count('\x8B'
↳ ')
        stat[i]=important_bytes
    sorted_stat = sorted(stat.iteritems(), key=operator.itemgetter(1),
↳ reverse=True)
    print sorted_stat[0]

```

(Исходный код можно скачать [здесь](#).)

Запускаю и вот ключ, для которого присутствие байт 0/0xFF/0x8B в дешифрованном буфере максимально:

```

N= 0
(147, 1224)
N= 1
(94, 1327)
N= 2
(252, 1223)
N= 3
(218, 1266)
N= 4

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
(38, 1209)
N= 5
(192, 1378)
N= 6
(199, 1204)
N= 7
(213, 1332)
N= 8
(225, 1251)
N= 9
(112, 1223)
N= 10
(143, 1177)
N= 11
(108, 1286)
N= 12
(10, 1164)
N= 13
(3, 1271)
N= 14
(128, 1253)
N= 15
(232, 1330)
```

Напишем утилиту для дешифрования для полученного ключа:

```
#!/usr/bin/env python
import sys, hexdump, array

def xor_strings(s,t):
    # https://en.wikipedia.org/wiki/XOR_cipher#Example_implementation
    """xor two strings together"""
    return "".join(chr(ord(a)^ord(b)) for a,b in zip(s,t))

IV=array.array('B', [147, 94, 252, 218, 38, 192, 199, 213, 225, 112, 143, ↵
↵ 108, 10, 3, 128, 232]).tostring()

def chunks(l, n):
    n = max(1, n)
    return [l[i:i + n] for i in range(0, len(l), n)]

def read_file(fname):
    file=open(fname, mode='rb')
    content=file.read()
    file.close()
    return content

def decrypt_byte(i, k):
    return chr ((ord(i)-k) % 256)

def decrypt(buf):
    return "".join(decrypt_byte(buf[i], i) for i in range(16))

fout=open(sys.argv[2], mode='wb')
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

prev=IV
content=read_file(sys.argv[1])
tmp=chunks(content, 16)
for c in tmp:
    new_c=decrypt(c)
    p=xor_strings (new_c, prev)
    prev=xor_strings(c, p)
    fout.write(p)
fout.close()

```

(Исходный код можно скачать [здесь](#).)

Проверим итоговый файл:

```

$ objdump -b binary -m i386 -D decrypted.bin

...

5:      8b ff          mov    %edi,%edi
7:      55            push  %ebp
8:      8b ec          mov    %esp,%ebp
a:      51            push  %ecx
b:      53            push  %ebx
c:      33 db          xor    %ebx,%ebx
e:      43            inc   %ebx
f:      84 1d a0 e2 05 01 test  %bl,0x105e2a0
15:     75 09          jne   0x20
17:     ff 75 08      pushl 0x8(%ebp)
1a:     ff 15 b0 13 00 01 call  *0x10013b0
20:     6a 6c          push  $0x6c
22:     ff 35 54 d0 01 01 pushl 0x101d054
28:     ff 15 b4 13 00 01 call  *0x10013b4
2e:     89 45 fc          mov    %eax,-0x4(%ebp)
31:     85 c0          test  %eax,%eax
33:     0f 84 d9 00 00 00 je    0x112
39:     56            push  %esi
3a:     57            push  %edi
3b:     6a 00          push  $0x0
3d:     50            push  %eax
3e:     ff 15 b8 13 00 01 call  *0x10013b8
44:     8b 35 bc 13 00 01 mov    0x10013bc,%esi
4a:     8b f8          mov    %eax,%edi
4c:     a1 e0 e2 05 01 mov    0x105e2e0,%eax
51:     3b 05 e4 e2 05 01 cmp   0x105e2e4,%eax
57:     75 12          jne   0x6b
59:     53            push  %ebx
5a:     6a 03          push  $0x3
5c:     57            push  %edi
5d:     ff d6          call  *%esi

...

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Да, выглядит как корректно дизассемблированный кусок x86-кода. Весь дешифрованный файл можно скачать [здесь](#).

На самом деле, это исполняемая секция из regedit.exe из Windows 7. Но этот пример основан на реальном случае из моей практики, так что только исполняемый файл другой (и ключ), а алгоритм тот же.

### 8.9.1. Еще идеи для рассмотрения

Что если бы не получилось с простым частотным анализом? Есть и другие идеи о том, как измерить корректность дешифрованного/разжатого x86-кода:

- Многие современные компиляторы выравнивают начало ф-ций по 16-байтной границе. Так что пространство оставленное перед ними заполняется NOP-ами (0x90) или иными NOP-инструкциями с известными опкодами: ?? (стр. ??). Либо же инструкциями INT3 (0xCC).
- Наверное, самый частый шаблонный код в ассемблере это вызов ф-ции: PUSH chain / CALL / ADD ESP, X. Эту последовательность легко обнаруживать и находить. Я даже собирал статистику о среднем количестве аргументов ф-ций: ?? (стр. ??). (Т.е., это средняя длина цепи PUSH-инструкций.)

Читайте еще о некорректно/корректно дизассемблированном коде: ?? (стр. ??).

## 8.10. SAP

### 8.10.1. Касательно сжимания сетевого трафика в клиенте SAP

(Эта статья в начале появилась в моем блоге, 13-июля-2010.)

(Трассировка связи между переменной окружения TDW\_NOCOMPRESS SAPGUI<sup>27</sup> до «назойливого всплывающего окна» и самой функции сжатия данных.)

Известно, что сетевой трафик между SAPGUI и SAP по умолчанию не шифруется, а сжимается

(читайте [здесь](#)<sup>28</sup> и [здесь](#)<sup>29</sup>).

Известно также что если установить переменную окружения TDW\_NOCOMPRESS в 1, можно выключить сжатие сетевых пакетов.

Но вы увидите окно, которое нельзя будет закрыть:

---

<sup>27</sup> GUI-клиент от SAP

<sup>28</sup> <http://blog.yurichev.com/node/44>

<sup>29</sup> [blog.yurichev.com](http://blog.yurichev.com)

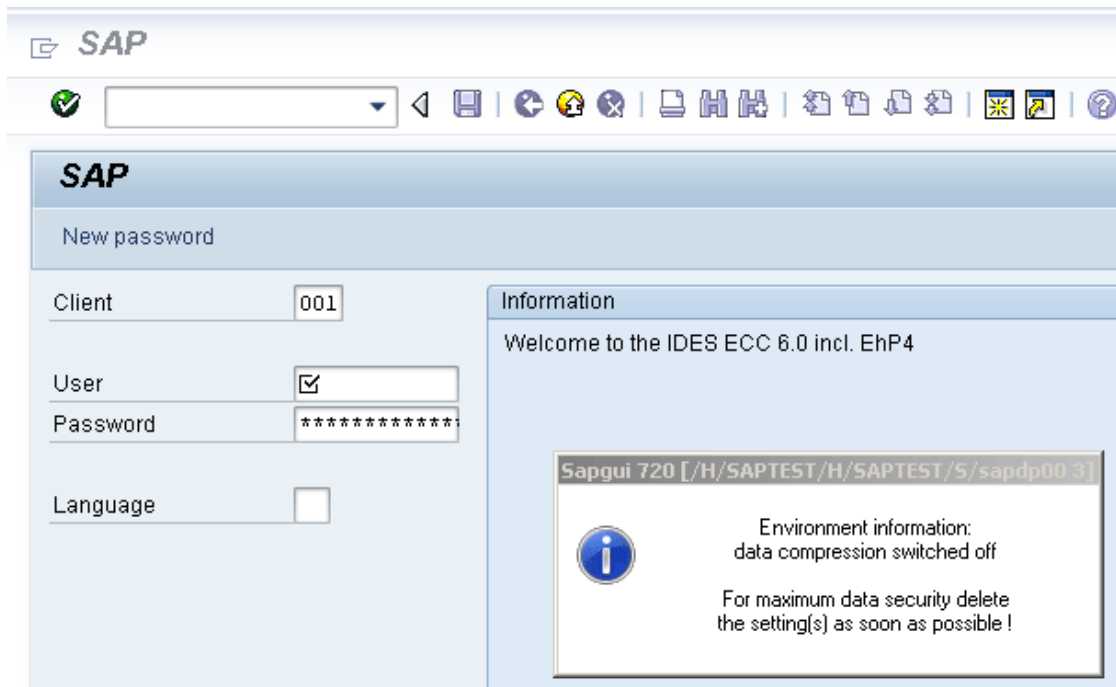


Рис. 8.17: Скриншот

Посмотрим, сможем ли мы как-то убрать это окно.

Но в начале давайте посмотрим, что мы уже знаем. Первое: мы знаем, что переменная окружения *TDW\_NOCOMPRESS* проверяется где-то внутри клиента SAPGUI.

Второе: строка вроде «data compression switched off» также должна где-то присутствовать.

При помощи файлового менеджера FAR<sup>30</sup> мы можем найти обе эти строки в файле SAPguilib.dll.

Так что давайте откроем файл SAPguilib.dll в **IDA!** и поищем там строку *TDW\_NOCOMPRESS*. Да, она присутствует и имеется только одна ссылка на эту строку.

Мы увидим такой фрагмент кода (все смещения верны для версии SAPGUI 720 win32, SAPguilib.dll версия файла 7200,1,0,9009):

.text:6440D51B	lea	eax, [ebp+2108h+var_211C]
.text:6440D51E	push	eax ; int
.text:6440D51F	push	offset aTdw_nocompress ;
"TDW_NOCOMPRESS"		
.text:6440D524	mov	byte ptr [edi+15h], 0
.text:6440D528	call	chk_env
.text:6440D52D	pop	ecx

<sup>30</sup><http://www.farmanager.com/>

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
.text:6440D52E      pop     ecx
.text:6440D52F      push   offset byte_64443AF8
.text:6440D534      lea    ecx, [ebp+2108h+var_211C]

; demangled name: int ATL::CStringT::Compare(char const *)const
.text:6440D537      call   ds:mfc90_1603
.text:6440D53D      test   eax, eax
.text:6440D53F      jz     short loc_6440D55A
.text:6440D541      lea    ecx, [ebp+2108h+var_211C]

; demangled name: const char* ATL::CStringT::operator PCWSTR
.text:6440D544      call   ds:mfc90_910
.text:6440D54A      push   eax                ; Str
.text:6440D54B      call   ds:atoi
.text:6440D551      test   eax, eax
.text:6440D553      setnz  al
.text:6440D556      pop    ecx
.text:6440D557      mov    [edi+15h], al
```

Строка возвращаемая функцией `chk_env()` через второй аргумент, обрабатывается далее строковыми функциями MFC, затем вызывается `atoi()`<sup>31</sup>. После этого, число сохраняется в `edi+15h`.

Обратите также внимание на функцию `chk_env` (это мы так назвали её вручную):

```
.text:64413F20 ; int __cdecl chk_env(char *VarName, int)
.text:64413F20 chk_env      proc near
.text:64413F20
.text:64413F20 DstSize      = dword ptr -0Ch
.text:64413F20 var_8         = dword ptr -8
.text:64413F20 DstBuf       = dword ptr -4
.text:64413F20 VarName      = dword ptr 8
.text:64413F20 arg_4        = dword ptr 0Ch
.text:64413F20
.text:64413F20      push   ebp
.text:64413F21      mov    ebp, esp
.text:64413F23      sub    esp, 0Ch
.text:64413F26      mov    [ebp+DstSize], 0
.text:64413F2D      mov    [ebp+DstBuf], 0
.text:64413F34      push  offset unk_6444C88C
.text:64413F39      mov    ecx, [ebp+arg_4]

; (demangled name) ATL::CStringT::operator=(char const *)
.text:64413F3C      call   ds:mfc90_820
.text:64413F42      mov    eax, [ebp+VarName]
.text:64413F45      push  eax                ; VarName
.text:64413F46      mov    ecx, [ebp+DstSize]
.text:64413F49      push  ecx                ; DstSize
.text:64413F4A      mov    edx, [ebp+DstBuf]
.text:64413F4D      push  edx                ; DstBuf
.text:64413F4E      lea   eax, [ebp+DstSize]
```

<sup>31</sup>Стандартная функция `atoi`, конвертирующая число в строке в число



```

.text:64413F51      push    eax                ; ReturnSize
.text:64413F52      call   ds:getenv_s
.text:64413F58      add    esp, 10h
.text:64413F5B      mov    [ebp+var_8], eax
.text:64413F5E      cmp    [ebp+var_8], 0
.text:64413F62      jz     short loc_64413F68
.text:64413F64      xor    eax, eax
.text:64413F66      jmp    short loc_64413FBC
.text:64413F68
.text:64413F68  loc_64413F68:
.text:64413F68      cmp    [ebp+DstSize], 0
.text:64413F6C      jnz    short loc_64413F72
.text:64413F6E      xor    eax, eax
.text:64413F70      jmp    short loc_64413FBC
.text:64413F72
.text:64413F72  loc_64413F72:
.text:64413F72      mov    ecx, [ebp+DstSize]
.text:64413F75      push  ecx
.text:64413F76      mov    ecx, [ebp+arg_4]

; demangled name: ATL::CStringT<char, 1>::Preallocate(int)
.text:64413F79      call   ds:mfc90_2691
.text:64413F7F      mov    [ebp+DstBuf], eax
.text:64413F82      mov    edx, [ebp+VarName]
.text:64413F85      push  edx                ; VarName
.text:64413F86      mov    eax, [ebp+DstSize]
.text:64413F89      push  eax                ; DstSize
.text:64413F8A      mov    ecx, [ebp+DstBuf]
.text:64413F8D      push  ecx                ; DstBuf
.text:64413F8E      lea   edx, [ebp+DstSize]
.text:64413F91      push  edx                ; ReturnSize
.text:64413F92      call   ds:getenv_s
.text:64413F98      add    esp, 10h
.text:64413F9B      mov    [ebp+var_8], eax
.text:64413F9E      push  0FFFFFFFFh
.text:64413FA0      mov    ecx, [ebp+arg_4]

; demangled name: ATL::CStringT::ReleaseBuffer(int)
.text:64413FA3      call   ds:mfc90_5835
.text:64413FA9      cmp    [ebp+var_8], 0
.text:64413FAD      jz     short loc_64413FB3
.text:64413FAF      xor    eax, eax
.text:64413FB1      jmp    short loc_64413FBC
.text:64413FB3
.text:64413FB3  loc_64413FB3:
.text:64413FB3      mov    ecx, [ebp+arg_4]

; demangled name: const char* ATL::CStringT::operator PCWSTR
.text:64413FB6      call   ds:mfc90_910
.text:64413FBC
.text:64413FBC  loc_64413FBC:
.text:64413FBC
.text:64413FBC      mov    esp, ebp

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

.text:64413FBE	pop	ebp
.text:64413FBF	retn	
.text:64413FBF chk_env	endp	

Да. Функция `getenv_s()`<sup>32</sup> это *безопасная* версия функции `getenv()`<sup>33</sup> в MSVC.

Тут также имеются манипуляции со строками при помощи функций из MFC.

Множество других переменных окружения также проверяются. Здесь список всех переменных проверяемых SAPGUI а также сообщение записываемое им в лог-файл, если переменная включена:

DPTRACE	"GUI-OPTION: Trace set to %d"
TDW_HEXDUMP	"GUI-OPTION: Hexdump enabled"
TDW_WORKDIR	"GUI-OPTION: working directory '%s'"
TDW_SPLASHSRCEENOFF	"GUI-OPTION: Splash Screen Off"
	"GUI-OPTION: Splash Screen On"
TDW_REPLYTIMEOUT	"GUI-OPTION: reply timeout %d milliseconds"
TDW_PLAYBACKTIMEOUT	"GUI-OPTION: PlaybackTimeout set to %d milliseconds"
TDW_NOCOMPRESS	"GUI-OPTION: no compression read"
TDW_EXPERT	"GUI-OPTION: expert mode"
TDW_PLAYBACKPROGRESS	"GUI-OPTION: PlaybackProgress"
TDW_PLAYBACKNETTRAFFIC	"GUI-OPTION: PlaybackNetTraffic"
TDW_PLAYLOG	"GUI-OPTION: /PlayLog is YES, file %s"
TDW_PLAYTIME	"GUI-OPTION: /PlayTime set to %d milliseconds"
TDW_LOGFILE	"GUI-OPTION: TDW_LOGFILE '%s'"
TDW_WAN	"GUI-OPTION: WAN - low speed connection enabled"
TDW_FULLMENU	"GUI-OPTION: FullMenu enabled"
SAP_CP / SAP_CODEPAGE	"GUI-OPTION: SAP_CODEPAGE '%d'"
UPDOWNLOAD_CP	"GUI-OPTION: UPDOWNLOAD_CP '%d'"
SNC_PARTNERNAME	"GUI-OPTION: SNC name '%s'"
SNC_QOP	"GUI-OPTION: SNC_QOP '%s'"
SNC_LIB	"GUI-OPTION: SNC is set to: %s"
SAPGUI_INPLACE	"GUI-OPTION: environment variable SAPGUI_INPLACE is on"

Настройки для каждой переменной записываются в массив через указатель в регистре EDI. EDI выставляется перед вызовом функции:

.text:6440EE00	lea	edi, [ebp+2884h+var_2884] ; options
here like +0x15...		
.text:6440EE03	lea	ecx, [esi+24h]
.text:6440EE06	call	load_command_line
.text:6440EE0B	mov	edi, eax
.text:6440EE0D	xor	ebx, ebx
.text:6440EE0F	cmp	edi, ebx
.text:6440EE11	jz	short loc_6440EE42
.text:6440EE13	push	edi
.text:6440EE14	push	offset aSapguiStoppedA ; "Sapgui
stopped after commandline interp"...		
.text:6440EE19	push	dword_644F93E8
.text:6440EE1F	call	FEWTraceError

<sup>32</sup>MSDN

<sup>33</sup>Стандартная функция Си, возвращающая значение переменной окружения

А теперь, можем ли мы найти строку *data record mode switched on*? Да, и есть только одна ссылка на эту строку в функции.

CDwsGui::PrepareInfoWindow(). Откуда мы узнали имена классов/методов? Здесь много специальных отладочных вызовов, пишущих в лог-файл вроде:

```
.text:64405160      push    dword ptr [esi+2854h]
.text:64405166      push    offset aCdwsGuiPrepare ;
                  "\nCDwsGui::PrepareInfoWindow: sapgui env"...
.text:6440516B      push    dword ptr [esi+2848h]
.text:64405171      call   dbg
.text:64405176      add     esp, 0Ch
```

...или:

```
.text:6440237A      push    eax
.text:6440237B      push    offset aCClientStart_6 ;
                  "CClient::Start: set shortcut user to '%'"...
.text:64402380      push    dword ptr [edi+4]
.text:64402383      call   dbg
.text:64402388      add     esp, 0Ch
```

Они *очень* полезны.

Посмотрим содержимое функции «назойливого всплывающего окна»:

```
.text:64404F4F CDwsGui__PrepareInfoWindow proc near
.text:64404F4F
.text:64404F4F pvParam      = byte ptr -3Ch
.text:64404F4F var_38        = dword ptr -38h
.text:64404F4F var_34        = dword ptr -34h
.text:64404F4F rc           = tagRECT ptr -2Ch
.text:64404F4F cy           = dword ptr -1Ch
.text:64404F4F h            = dword ptr -18h
.text:64404F4F var_14        = dword ptr -14h
.text:64404F4F var_10        = dword ptr -10h
.text:64404F4F var_4         = dword ptr -4
.text:64404F4F
.text:64404F4F      push    30h
.text:64404F51      mov     eax, offset loc_64438E00
.text:64404F56      call   __EH_prolog3
.text:64404F5B      mov     esi, ecx          ; ECX is pointer to
                        object
.text:64404F5D      xor     ebx, ebx
.text:64404F5F      lea    ecx, [ebp+var_14]
.text:64404F62      mov     [ebp+var_10], ebx

; demangled name: ATL::CStringT(void)
.text:64404F65      call   ds:mfc90_316
.text:64404F6B      mov     [ebp+var_4], ebx
.text:64404F6E      lea    edi, [esi+2854h]
.text:64404F74      push    offset aEnvironmentInf ;
                  "Environment information:\n"
.text:64404F79      mov     ecx, edi
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; demangled name: ATL::CStringT::operator=(char const *)
.text:64404F7B      call     ds:mfc90_820
.text:64404F81      cmp     [esi+38h], ebx
.text:64404F84      mov     ebx, ds:mfc90_2539
.text:64404F8A      jbe    short loc_64404FA9
.text:64404F8C      push   dword ptr [esi+34h]
.text:64404F8F      lea    eax, [ebp+var_14]
.text:64404F92      push   offset aWorkingDirecto ;
        "working directory: '%s'\n"
.text:64404F97      push   eax

; demangled name: ATL::CStringT::Format(char const *,...)
.text:64404F98      call   ebx ; mfc90_2539
.text:64404F9A      add    esp, 0Ch
.text:64404F9D      lea    eax, [ebp+var_14]
.text:64404FA0      push   eax
.text:64404FA1      mov    ecx, edi

;
        demangled name: ATL::CStringT::operator+=(class ATL::CStringT<char, 1> const &)
.text:64404FA3      call   ds:mfc90_941
.text:64404FA9
.text:64404FA9  loc_64404FA9:
.text:64404FA9      mov    eax, [esi+38h]
.text:64404FAC      test   eax, eax
.text:64404FAE      jbe    short loc_64404FD3
.text:64404FB0      push   eax
.text:64404FB1      lea    eax, [ebp+var_14]
.text:64404FB4      push   offset aTraceLevelDAct ;
        "trace level %d activated\n"
.text:64404FB9      push   eax

; demangled name: ATL::CStringT::Format(char const *,...)
.text:64404FBA      call   ebx ; mfc90_2539
.text:64404FBC      add    esp, 0Ch
.text:64404FBF      lea    eax, [ebp+var_14]
.text:64404FC2      push   eax
.text:64404FC3      mov    ecx, edi

;
        demangled name: ATL::CStringT::operator+=(class ATL::CStringT<char, 1> const &)
.text:64404FC5      call   ds:mfc90_941
.text:64404FCB      xor    ebx, ebx
.text:64404FCD      inc    ebx
.text:64404FCE      mov    [ebp+var_10], ebx
.text:64404FD1      jmp    short loc_64404FD6
.text:64404FD3
.text:64404FD3  loc_64404FD3:
.text:64404FD3      xor    ebx, ebx
.text:64404FD5      inc    ebx
.text:64404FD6
.text:64404FD6  loc_64404FD6:
.text:64404FD6      cmp    [esi+38h], ebx
.text:64404FD9      jbe    short loc_64404FF1

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:64404FDB      cmp     dword ptr [esi+2978h], 0
.text:64404FE2      jz      short loc_64404FF1
.text:64404FE4      push   offset aHexdumpInTrace ;
                "hexdump in trace activated\n"
.text:64404FE9      mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64404FEB      call   ds:mfc90_945
.text:64404FF1
.text:64404FF1 loc_64404FF1:
.text:64404FF1
.text:64404FF1      cmp     byte ptr [esi+78h], 0
.text:64404FF5      jz      short loc_64405007
.text:64404FF7      push   offset aLoggingActivat ;
                "logging activated\n"
.text:64404FFC      mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64404FFE      call   ds:mfc90_945
.text:64405004      mov     [ebp+var_10], ebx
.text:64405007
.text:64405007 loc_64405007:
.text:64405007      cmp     byte ptr [esi+3Dh], 0
.text:6440500B      jz      short bypass
.text:6440500D      push   offset aDataCompressio ;
                "data compression switched off\n"
.text:64405012      mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405014      call   ds:mfc90_945
.text:6440501A      mov     [ebp+var_10], ebx
.text:6440501D
.text:6440501D bypass:
.text:6440501D      mov     eax, [esi+20h]
.text:64405020      test   eax, eax
.text:64405022      jz      short loc_6440503A
.text:64405024      cmp     dword ptr [eax+28h], 0
.text:64405028      jz      short loc_6440503A
.text:6440502A      push   offset aDataRecordMode ;
                "data record mode switched on\n"
.text:6440502F      mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405031      call   ds:mfc90_945
.text:64405037      mov     [ebp+var_10], ebx
.text:6440503A
.text:6440503A loc_6440503A:
.text:6440503A
.text:6440503A      mov     ecx, edi
.text:6440503C      cmp     [ebp+var_10], ebx
.text:6440503F      jnz    loc_64405142
.text:64405045      push   offset aForMaximumData ;
                "\nFor maximum data security delete\nthe s"...

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; demangled name: ATL::CStringT::operator+=(char const *)
.text:6440504A      call     ds:mfc90_945
.text:64405050      xor     edi, edi
.text:64405052      push    edi                ; fWinIni
.text:64405053      lea    eax, [ebp+pvParam]
.text:64405056      push    eax                ; pvParam
.text:64405057      push    edi                ; uiParam
.text:64405058      push    30h                ; uiAction
.text:6440505A      call   ds:SystemParametersInfoA
.text:64405060      mov    eax, [ebp+var_34]
.text:64405063      cmp    eax, 1600
.text:64405068      jle    short loc_64405072
.text:6440506A      cdq
.text:6440506B      sub    eax, edx
.text:6440506D      sar    eax, 1
.text:6440506F      mov    [ebp+var_34], eax
.text:64405072
.text:64405072  loc_64405072:
.text:64405072      push    edi                ; hWnd
.text:64405073      mov    [ebp+cy], 0A0h
.text:6440507A      call   ds:GetDC
.text:64405080      mov    [ebp+var_10], eax
.text:64405083      mov    ebx, 12Ch
.text:64405088      cmp    eax, edi
.text:6440508A      jz     loc_64405113
.text:64405090      push    11h                ; i
.text:64405092      call   ds:GetStockObject
.text:64405098      mov    edi, ds>SelectObject
.text:6440509E      push    eax                ; h
.text:6440509F      push    [ebp+var_10]        ; hdc
.text:644050A2      call   edi ; SelectObject
.text:644050A4      and    [ebp+rc.left], 0
.text:644050A8      and    [ebp+rc.top], 0
.text:644050AC      mov    [ebp+h], eax
.text:644050AF      push    401h                ; format
.text:644050B4      lea    eax, [ebp+rc]
.text:644050B7      push    eax                ; lprc
.text:644050B8      lea    ecx, [esi+2854h]
.text:644050BE      mov    [ebp+rc.right], ebx
.text:644050C1      mov    [ebp+rc.bottom], 0B4h

; demangled name: ATL::CStringT::GetLength(void)
.text:644050C8      call   ds:mfc90_3178
.text:644050CE      push    eax                ; cchText
.text:644050CF      lea    ecx, [esi+2854h]

; demangled name: const char* ATL::CStringT::operator PCWSTR
.text:644050D5      call   ds:mfc90_910
.text:644050DB      push    eax                ; lpchText
.text:644050DC      push    [ebp+var_10]        ; hdc
.text:644050DF      call   ds:DrawTextA
.text:644050E5      push    4                  ; nIndex
.text:644050E7      call   ds:GetSystemMetrics

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:644050ED      mov     ecx, [ebp+rc.bottom]
.text:644050F0      sub     ecx, [ebp+rc.top]
.text:644050F3      cmp     [ebp+h], 0
.text:644050F7      lea    eax, [eax+ecx+28h]
.text:644050FB      mov     [ebp+cy], eax
.text:644050FE      jz     short loc_64405108
.text:64405100      push   [ebp+h]          ; h
.text:64405103      push   [ebp+var_10]     ; hdc
.text:64405106      call   edi ; SelectObject
.text:64405108      loc_64405108:
.text:64405108      push   [ebp+var_10]     ; hdc
.text:6440510B      push   0                ; hWnd
.text:6440510D      call   ds:ReleaseDC
.text:64405113      loc_64405113:
.text:64405113      mov     eax, [ebp+var_38]
.text:64405116      push   80h              ; uFlags
.text:6440511B      push   [ebp+cy]         ; cy
.text:6440511E      inc    eax
.text:6440511F      push   ebx              ; cx
.text:64405120      push   eax              ; Y
.text:64405121      mov     eax, [ebp+var_34]
.text:64405124      add    eax, 0FFFFFFD4h
.text:64405129      cdq
.text:6440512A      sub    eax, edx
.text:6440512C      sar    eax, 1
.text:6440512E      push   eax              ; X
.text:6440512F      push   0                ; hWndInsertAfter
.text:64405131      push   dword ptr [esi+285Ch] ; hWnd
.text:64405137      call   ds:SetWindowPos
.text:6440513D      xor    ebx, ebx
.text:6440513F      inc    ebx
.text:64405140      jmp    short loc_6440514D
.text:64405142      loc_64405142:
.text:64405142      push   offset byte_64443AF8

; demangled name: ATL::CStringT::operator=(char const *)
.text:64405147      call   ds:mfc90_820
.text:6440514D      loc_6440514D:
.text:6440514D      cmp    dword_6450B970, ebx
.text:64405153      jl    short loc_64405188
.text:64405155      call   sub_6441C910
.text:6440515A      mov    dword_644F858C, ebx
.text:64405160      push   dword ptr [esi+2854h]
.text:64405166      push   offset aCDwsguiPrepare ;
"\nCDwsgui::PrepareInfoWindow: sapgui env"...
.text:6440516B      push   dword ptr [esi+2848h]
.text:64405171      call   dbg
.text:64405176      add    esp, 0Ch
.text:64405179      mov    dword_644F858C, 2
.text:64405183      call   sub_6441C920

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
.text:64405188
.text:64405188 loc_64405188:
.text:64405188         or      [ebp+var_4], 0FFFFFFFh
.text:6440518C         lea   ecx, [ebp+var_14]

; demangled name: ATL::CStringT:: CStringT()
.text:6440518F         call  ds:mfc90_601
.text:64405195         call  __EH_epilog3
.text:6440519A         retn
.text:6440519A CDwsgui__PrepareInfoWindow endp
```

ECX в начале функции содержит в себе указатель на объект (потому что это тип функции thiscall (?? (стр. ??))). В нашем случае, класс имеет тип, очевидно, *CDwsgui*. В зависимости от включенных опций в объекте, разные сообщения добавляются к итоговому сообщению.

Если переменная по адресу `this+0x3D` не ноль, компрессия сетевых пакетов будет выключена:

```
.text:64405007 loc_64405007:
.text:64405007         cmp   byte ptr [esi+3Dh], 0
.text:6440500B         jz    short bypass
.text:6440500D         push  offset aDataCompressio ;
        "data compression switched off\n"
.text:64405012         mov   ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405014         call  ds:mfc90_945
.text:6440501A         mov   [ebp+var_10], ebx
.text:6440501D
.text:6440501D bypass:
```

Интересно, что в итоге, состояние переменной `var_10` определяет, будет ли показано сообщение вообще:

```
.text:6440503C         cmp   [ebp+var_10], ebx
.text:6440503F         jnz   exit ; пропустить отрисовку

; добавить строки "For maximum data security delete" / "the setting(s) as
        soon as possible !":

.text:64405045         push  offset aForMaximumData ;
        "\nFor maximum data security delete\nthe s"...
.text:6440504A         call  ds:mfc90_945 ;
        ATL::CStringT::operator+=(char const *)
.text:64405050         xor   edi, edi
.text:64405052         push  edi ; fWinIni
.text:64405053         lea  eax, [ebp+pvParam]
.text:64405056         push  eax ; pvParam
.text:64405057         push  edi ; uiParam
.text:64405058         push  30h ; uiAction
.text:6440505A         call  ds:SystemParametersInfoA
.text:64405060         mov  eax, [ebp+var_34]
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

.text:64405063      cmp     eax, 1600
.text:64405068      jle     short loc_64405072
.text:6440506A      cdq
.text:6440506B      sub     eax, edx
.text:6440506D      sar     eax, 1
.text:6440506F      mov     [ebp+var_34], eax
.text:64405072
.text:64405072  loc_64405072:

; начинает рисовать:

.text:64405072      push    edi                ; hWnd
.text:64405073      mov     [ebp+cy], 0A0h
.text:6440507A      call   ds:GetDC

```

Давайте проверим нашу теорию на практике.

JNZ в этой строке ...

```

.text:6440503F      jnz     exit ; пропустить отрисовку

```

...заменяем просто на JMP и получим SAPGUI работающим без этого назойливого всплывающего окна!

Копнем немного глубже и проследим связь между смещением 0x15 в `load_command_line()` (Это мы дали имя этой функции) и переменной `this+0x3D` в `CDwsGui::PrepareInfoWindow`. Уверены ли мы что это одна и та же переменная?

Начинаем искать все места где в коде используется константа 0x15. Для таких небольших программ как SAPGUI, это иногда срабатывает. Вот первое что находим:

```

.text:64404C19  sub_64404C19  proc near
.text:64404C19
.text:64404C19  arg_0        = dword ptr 4
.text:64404C19
.text:64404C19      push    ebx
.text:64404C1A      push    ebp
.text:64404C1B      push    esi
.text:64404C1C      push    edi
.text:64404C1D      mov     edi, [esp+10h+arg_0]
.text:64404C21      mov     eax, [edi]
.text:64404C23      mov     esi, ecx ; ESI/ECX are pointers to
                 some unknown object.
.text:64404C25      mov     [esi], eax
.text:64404C27      mov     eax, [edi+4]
.text:64404C2A      mov     [esi+4], eax
.text:64404C2D      mov     eax, [edi+8]
.text:64404C30      mov     [esi+8], eax
.text:64404C33      lea    eax, [edi+0Ch]
.text:64404C36      push    eax
.text:64404C37      lea    ecx, [esi+0Ch]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; demangled name: ATL::CStringT::operator=(class ATL::CStringT ... &)
.text:64404C3A      call     ds:mfc90_817
.text:64404C40      mov     eax, [edi+10h]
.text:64404C43      mov     [esi+10h], eax
.text:64404C46      mov     al, [edi+14h]
.text:64404C49      mov     [esi+14h], al
.text:64404C4C      mov     al, [edi+15h] ; copy byte from 0x15
                    offset
.text:64404C4F      mov     [esi+15h], al ; to 0x15 offset in
                    CDwsGui object

```

Эта функция вызывается из функции с названием *CDwsGui::CopyOptions!* И снова спасибо отладочной информации.

Но настоящий ответ находится в функции *CDwsGui::Init()*:

```

.text:6440B0BF  loc_6440B0BF:
.text:6440B0BF      mov     eax, [ebp+arg_0]
.text:6440B0C2      push   [ebp+arg_4]
.text:6440B0C5      mov     [esi+2844h], eax
.text:6440B0CB      lea    eax, [esi+28h] ; ESI is pointer to
                    CDwsGui object
.text:6440B0CE      push   eax
.text:6440B0CF      call   CDwsGui__CopyOptions

```

Теперь ясно: массив заполняемый в *load\_command\_line()* на самом деле расположен в классе *CDwsGui* но по адресу *this+0x28*.  $0x15 + 0x28$  это  $0x3D$ . ОК, мы нашли место, куда наша переменная копируется.

Посмотрим также и другие места, где используется смещение  $0x3D$ . Одно из таких мест находится в функции *CDwsGui::SapguiRun* (и снова спасибо отладочным вызовам):

```

.text:64409D58      cmp     [esi+3Dh], bl ; ESI is pointer to
                    CDwsGui object
.text:64409D5B      lea    ecx, [esi+2B8h]
.text:64409D61      setz   al
.text:64409D64      push   eax ; arg_10 of
                    CConnectionContext::CreateNetwork
.text:64409D65      push   dword ptr [esi+64h]

; demangled name: const char* ATL::CStringT::operator PCWSTR
.text:64409D68      call   ds:mfc90_910
                    ; no arguments
.text:64409D6E      push   eax
.text:64409D6F      lea    ecx, [esi+2BCh]

; demangled name: const char* ATL::CStringT::operator PCWSTR
.text:64409D75      call   ds:mfc90_910
                    ; no arguments
.text:64409D7B      push   eax
.text:64409D7C      push   esi
.text:64409D7D      lea    ecx, [esi+8]
.text:64409D80      call   CConnectionContext__CreateNetwork

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Проверим нашу идею.

Заменяем `setz al` здесь на `xor eax, eax / nop`, убираем переменную окружения `TDW_NOCOMPRESS` и запускаем `SAPGUI`. Ух! Назойливого окна больше нет (как и ожидалось: ведь переменной окружения также нет), но в `Wireshark` мы видим, что сетевые пакеты больше не сжимаются! Очевидно, это то самое место где флаг отражающий сжатие пакетов выставляется в объекте `CConnectionContext`.

Так что, флаг сжатия передается в пятом аргументе функции `CConnectionContext::CreateNetwork`. Внутри этой функции, вызывается еще одна:

```

...
.text:64403476      push    [ebp+compression]
.text:64403479      push    [ebp+arg_C]
.text:6440347C      push    [ebp+arg_8]
.text:6440347F      push    [ebp+arg_4]
.text:64403482      push    [ebp+arg_0]
.text:64403485      call   CNetwork__CNetwork

```

Флаг отвечающий за сжатие здесь передается в пятом аргументе для конструктора `CNetwork::CNetwork`.

И вот как конструктор `CNetwork` выставляет некоторые флаги в объекте `CNetwork` в соответствии с пятым аргументом *и* еще какую-то переменную, возможно, также отвечающую за сжатие сетевых пакетов.

```

.text:64411DF1      cmp     [ebp+compression], esi
.text:64411DF7      jz     short set_EAX_to_0
.text:64411DF9      mov     al, [ebx+78h] ; another value may
affect compression?
.text:64411DFC      cmp     al, '3'
.text:64411DFE      jz     short set_EAX_to_1
.text:64411E00      cmp     al, '4'
.text:64411E02      jnz    short set_EAX_to_0
.text:64411E04      set_EAX_to_1:
.text:64411E04      xor     eax, eax
.text:64411E06      inc     eax ; EAX -> 1
.text:64411E07      jmp     short loc_64411E0B
.text:64411E09      set_EAX_to_0:
.text:64411E09      xor     eax, eax ; EAX -> 0
.text:64411E0B      loc_64411E0B:
.text:64411E0B      mov     [ebx+3A4h], eax ; EBX is pointer to
CNetwork object

```

Теперь мы знаем, что флаг отражающий сжатие данных сохраняется в классе `CNetwork` по адресу `this+0x3A4`.

Поискем теперь значение `0x3A4` в `SAPguilib.dll`. Находим второе упоминание этого значения в функции `CDwsGui::OnClientMessageWrite` (бесконечная благодарность отладочной информации):

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.text:64406F76 loc_64406F76:
.text:64406F76      mov     ecx, [ebp+7728h+var_7794]
.text:64406F79      cmp     dword ptr [ecx+3A4h], 1
.text:64406F80      jnz    compression_flag_is_zero
.text:64406F86      mov     byte ptr [ebx+7], 1
.text:64406F8A      mov     eax, [esi+18h]
.text:64406F8D      mov     ecx, eax
.text:64406F8F      test   eax, eax
.text:64406F91      ja     short loc_64406FFF
.text:64406F93      mov     ecx, [esi+14h]
.text:64406F96      mov     eax, [esi+20h]
.text:64406F99
.text:64406F99 loc_64406F99:
.text:64406F99      push   dword ptr [edi+2868h] ; int
.text:64406F9F      lea   edx, [ebp+7728h+var_77A4]
.text:64406FA2      push   edx ; int
.text:64406FA3      push   30000 ; int
.text:64406FA8      lea   edx, [ebp+7728h+Dst]
.text:64406FAB      push   edx ; Dst
.text:64406FAC      push   ecx ; int
.text:64406FAD      push   eax ; Src
.text:64406FAE      push   dword ptr [edi+28C0h] ; int
.text:64406FB4      call  sub_644055C5 ; actual
      compression routine
.text:64406FB9      add   esp, 1Ch
.text:64406FBC      cmp   eax, 0FFFFFFFh
.text:64406FBF      jz    short loc_64407004
.text:64406FC1      cmp   eax, 1
.text:64406FC4      jz    loc_6440708C
.text:64406FCA      cmp   eax, 2
.text:64406FCD      jz    short loc_64407004
.text:64406FCF      push  eax
.text:64406FD0      push  offset aCompressionErr ;
      "compression error [rc = %d]- program wi"...
.text:64406FD5      push  offset aGui_err_compre ;
      "GUI_ERR_COMPRESS"
.text:64406FDA      push  dword ptr [edi+28D0h]
.text:64406FE0      call  SapPcTxtRead

```

Заглянем в функцию `sub_644055C5`. Всё что в ней мы находим это вызов `metcsru()` и еще какую-то функцию, названную **IDA!** `sub_64417440`.

И теперь заглянем в `sub_64417440`. Увидим там:

```

.text:6441747C      push   offset aErrorCsRcompre ;
      "\nERROR: CsRCompress: invalid handle"
.text:64417481      call  eax ; dword_644F94C8
.text:64417483      add   esp, 4

```

Voilà! Мы находим функцию которая собственно и сжимает сетевые пакеты. Как уже было видно в прошлом <sup>34</sup>, эта функция используется в SAP и в опенсорсном проекте MaxDB. Так что эта функция доступна в виде исходников.

<sup>34</sup>[http://conus.info/utills/SAP\\_pkt\\_decompr.txt](http://conus.info/utills/SAP_pkt_decompr.txt)

Последняя проверка:

.text:64406F79	cmp	dword ptr [ecx+3A4h], 1
.text:64406F80	jnz	compression_flag_is_zero

Заменяем JNZ на безусловный переход JMP. Уберем переменную окружения TDW\_NOCOMPRESS. Вуаля!

В Wireshark мы видим, что сетевые пакеты, исходящие от клиента, не сжаты. Ответы сервера, впрочем, сжаты.

Так что мы нашли связь между переменной окружения и местом где функция сжатия данных вызывается, а также может быть отключена.

### 8.10.2. Функции проверки пароля в SAP 6.0

Когда автор этой книги в очередной раз вернулся к своему SAP 6.0 IDES заинсталлированному в виртуальной машине VMware, он обнаружил что забыл пароль, впрочем, затем он вспомнил его, но теперь получаем такую ошибку:

«*Password logon no longer possible - too many failed attempts*», потому что были потрачены все попытки на то, чтобы вспомнить его.

Первая очень хорошая новость состоит в том, что с SAP поставляется полный PDB-файл *disp+work.pdb*, он содержит все: имена функций, структуры, типы, локальные переменные, имена аргументов, и т.д. Какой щедрый подарок!

Существует утилита TYPEINFODUMP<sup>35</sup> для дампа содержимого PDB-файлов во что-то более читаемое и грег-абельное.

Вот пример её работы: информация о функции + её аргументах + её локальных переменных:

```

FUNCTION ThVmcSysEvent
  Address:      10143190  Size:      675 bytes  Index:      60483  ↗
  ↳ TypeIndex:   60484
  Type: int NEAR_C ThVmcSysEvent (unsigned int, unsigned char, unsigned ↗
  ↳ short*)
Flags: 0
PARAMETER events
  Address: Reg335+288  Size:      4 bytes  Index:      60488  TypeIndex:   ↗
  ↳ 60489
  Type: unsigned int
Flags: d0
PARAMETER opcode
  Address: Reg335+296  Size:      1 bytes  Index:      60490  TypeIndex:   ↗
  ↳ 60491
  Type: unsigned char
Flags: d0
PARAMETER serverName
  Address: Reg335+304  Size:      8 bytes  Index:      60492  TypeIndex:   ↗
  ↳ 60493
  Type: unsigned short*

```

<sup>35</sup><http://www.debuginfo.com/tools/typeinfodump.html>

```

Flags: d0
STATIC_LOCAL_VAR func
  Address:      12274af0  Size:      8 bytes  Index:      60495 ↗
    ↳ TypeIndex: 60496
  Type: wchar_t*
Flags: 80
LOCAL_VAR admhead
  Address: Reg335+304  Size:      8 bytes  Index:      60498  TypeIndex: ↗
    ↳ 60499
  Type: unsigned char*
Flags: 90
LOCAL_VAR record
  Address: Reg335+64  Size:     204 bytes  Index:      60501  TypeIndex: ↗
    ↳ 60502
  Type: AD_RECORD
Flags: 90
LOCAL_VAR adlen
  Address: Reg335+296  Size:      4 bytes  Index:      60508  TypeIndex: ↗
    ↳ 60509
  Type: int
Flags: 90

```

А вот пример дампа структуры:

```

STRUCT DBSL_STMTID
Size: 120 Variables: 4 Functions: 0 Base classes: 0
MEMBER moduletype
  Type: DBSL_MODULETYPE
  Offset: 0 Index: 3 TypeIndex: 38653
MEMBER module
  Type: wchar_t module[40]
  Offset: 4 Index: 3 TypeIndex: 831
MEMBER stmtnum
  Type: long
  Offset: 84 Index: 3 TypeIndex: 440
MEMBER timestamp
  Type: wchar_t timestamp[15]
  Offset: 88 Index: 3 TypeIndex: 6612

```

Bay!

Вторая хорошая новость: *отладочные* вызовы, коих здесь очень много, очень полезны.

Здесь вы можете увидеть глобальную переменную `ct_level`<sup>36</sup>, отражающую уровень трассировки.

В `disp+work.exe` очень много таких отладочных вставок:

```

cmp     cs:ct_level, 1
jl      short loc_1400375DA
call    DpLock

```

<sup>36</sup>Еще об уровне трассировки: [http://help.sap.com/saphelp\\_nwpi71/helpdata/en/46/962416a5a613e8e1000000a155369/content.htm](http://help.sap.com/saphelp_nwpi71/helpdata/en/46/962416a5a613e8e1000000a155369/content.htm)

```

lea    rcx, aDpxxtool4_c ; "dpxxtool4.c"
mov    edx, 4Eh          ; line
call   CTrcSaveLocation
mov    r8, cs:func_48
mov    rcx, cs:hdl      ; hdl
lea    rdx, aSDpreadmemvalu ; "%s: DpReadMemValue (%d)"
mov    r9d, ebx
call   DpTrcErr
call   DpUnlock

```

Если текущий уровень трассировки выше или равен заданному в этом коде порогу, отладочное сообщение будет записано в лог-файл вроде *dev\_w0*, *dev\_disp* и прочие файлы *dev\**.

Попробуем грег-ать файл недавно полученный при помощи утилиты TYPEINFODUMP:

```
cat "disp+work.pdb.d" | grep FUNCTION | grep -i password
```

Получаем:

```

FUNCTION rcui::AgiPassword::DiagISelection
FUNCTION ssf_password_encrypt
FUNCTION ssf_password_decrypt
FUNCTION password_logon_disabled
FUNCTION dySignSkipUserPassword
FUNCTION migrate_password_history
FUNCTION password_is_initial
FUNCTION rcui::AgiPassword::IsVisible
FUNCTION password_distance_ok
FUNCTION get_password_downwards_compatibility
FUNCTION dySignUnSkipUserPassword
FUNCTION rcui::AgiPassword::GetTypeNames
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$2
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$0
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$1
FUNCTION usm_set_password
FUNCTION rcui::AgiPassword::TraceTo
FUNCTION days_since_last_password_change
FUNCTION rsecgrp_generate_random_password
FUNCTION rcui::AgiPassword::`scalar deleting destructor'
FUNCTION password_attempt_limit_exceeded
FUNCTION handle_incorrect_password
FUNCTION `rcui::AgiPassword::`scalar deleting destructor'::`1'::dtor$1
FUNCTION calculate_new_password_hash
FUNCTION shift_password_to_history
FUNCTION rcui::AgiPassword::GetType
FUNCTION found_password_in_history
FUNCTION `rcui::AgiPassword::`scalar deleting destructor'::`1'::dtor$0
FUNCTION rcui::AgiObj::IsaPassword
FUNCTION password_idle_check
FUNCTION SlicHwPasswordForDay
FUNCTION rcui::AgiPassword::IsaPassword
FUNCTION rcui::AgiPassword::AgiPassword
FUNCTION delete_user_password

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

FUNCTION usm_set_user_password
FUNCTION Password_API
FUNCTION get_password_change_for_SSO
FUNCTION password_in_USR40
FUNCTION rsec_agrp_abap_generate_random_password

```

Попробуем так же искать отладочные сообщения содержащие слова «password» и «locked». Одна из таких это строка «user was locked by subsequently failed password logon attempts» на которую есть ссылка в функции `password_attempt_limit_exceeded()`.

Другие строки, которые эта найденная функция может писать в лог-файл это: «password logon attempt will be rejected immediately (preventing dictionary attacks)», «failed-logon lock: expired (but not removed due to 'read-only' operation)», «failed-logon lock: expired => removed».

Немного поэкспериментировав с этой функцией, мы быстро понимаем, что проблема именно в ней. Она вызывается из функции `chckpass()` — одна из функций проверяющих пароль.

В начале, давайте убедимся, что мы на верном пути:

Запускаем `tracer`:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!chckpass,args:3,unicode
```

```

PID=2236|TID=2248|(0) disp+work.exe!chckpass (0x202c770, L"Brewered1
    ↙
    ↘ ", 0x41) (called from 0x1402f1060 (disp+work.
    ↙ exe!usrexist+0x3c0))
PID=2236|TID=2248|(0) disp+work.exe!chckpass -> 0x35

```

Функции вызываются так: `syssigni()` -> `DylSigni()` -> `dychkusr()` -> `usrexist()` -> `chckpass()`.

Число 0x35 возвращается из `chckpass()` в этом месте:

```

.text:00000001402ED567 loc_1402ED567: ; CODE XREF:
        chckpass+B4
.text:00000001402ED567 mov     rcx, rbx ; usr02
.text:00000001402ED56A call   password_idle_check
.text:00000001402ED56F cmp    eax, 33h
.text:00000001402ED572 jz     loc_1402EDB4E
.text:00000001402ED578 cmp    eax, 36h
.text:00000001402ED57B jz     loc_1402EDB3D
.text:00000001402ED581 xor    edx, edx ;
        usr02_readonly
.text:00000001402ED583 mov    rcx, rbx ; usr02
.text:00000001402ED586 call   ↗
        ↙ password_attempt_limit_exceeded
.text:00000001402ED58B test   al, al
.text:00000001402ED58D jz     short loc_1402ED5A0
.text:00000001402ED58F mov    eax, 35h
.text:00000001402ED594 add    rsp, 60h
.text:00000001402ED598 pop    r14

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```
.text:00000001402ED59A      pop     r12
.text:00000001402ED59C      pop     rdi
.text:00000001402ED59D      pop     rsi
.text:00000001402ED59E      pop     rbx
.text:00000001402ED59F      retn
```

Отлично, давайте проверим:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!↵
↳ password_attempt_limit_exceeded, args:4, unicode, rt:0
```

```
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded (0↵
↳ x202c770, 0, 0x257758, 0) (called from 0x1402ed58b (disp+work.exe!↵
↳ chckpass+0xeb))
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded -> 1
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded (0↵
↳ x202c770, 0, 0, 0) (called from 0x1402e9794 (disp+work.exe!chngpas+0↵
↳ xe4))
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded -> 1
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
```

Великолепно! Теперь мы можем успешно залогиниться.

Кстати, мы можем сделать вид что вообще забыли пароль, заставляя *chckpass()* всегда возвращать ноль, и этого достаточно для отключения проверки пароля:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!chckpass, args:3, unicode, rt↵
↳ :0
```

```
PID=2744|TID=360|(0) disp+work.exe!chckpass (0x202c770, L"bogus ↵
↳ ", 0x41) (called from 0x1402f1060 (disp+work.↵
↳ exe!usreexist+0x3c0))
PID=2744|TID=360|(0) disp+work.exe!chckpass -> 0x35
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
```

Что еще можно сказать, бегло анализируя функцию *password\_attempt\_limit\_exceeded()*, это то, что в начале можно увидеть следующий вызов:

```
lea     rcx, aLoginFailed_us ; "login/failed_user_auto_unlock"
call   sapparam
test   rax, rax
jz     short loc_1402E19DE
movzx  eax, word ptr [rax]
cmp    ax, 'N'
jz     short loc_1402E19D4
cmp    ax, 'n'
jz     short loc_1402E19D4
cmp    ax, '0'
jnz    short loc_1402E19DE
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Очевидно, функция *sapparam()* используется чтобы узнать значение какой-либо переменной конфигурации. Эта функция может вызываться из 1768 разных мест.

Вероятно, при помощи этой информации, мы можем легко находить те места кода, на которые влияют определенные переменные конфигурации.

Замечательно! Имена функций очень понятны, куда понятнее чем в Oracle RDBMS.

По всей видимости, процесс *disp+work* весь написан на Си++. Должно быть, он был переписан не так давно?

## 8.11. Oracle RDBMS

### 8.11.1. Таблица V\$VERSION в Oracle RDBMS

Oracle RDBMS 11.2 это очень большая программа, основной модуль *oracle.exe* содержит около 124 тысячи функций. Для сравнения, ядро Windows 7 x64 (*ntoskrnl.exe*) — около 11 тысяч функций, а ядро Linux 3.9.8 (с драйверами по умолчанию) — 31 тысяч функций.

Начнем с одного простого вопроса. Откуда Oracle RDBMS берет информацию, когда мы в SQL\*Plus пишем вот такой вот простой запрос:

```
SQL> select * from V$VERSION;
```

И получаем:

```
BANNER
-----
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
PL/SQL Release 11.2.0.1.0 - Production
CORE      11.2.0.1.0      Production
TNS for 32-bit Windows: Version 11.2.0.1.0 - Production
NLSRTL Version 11.2.0.1.0 - Production
```

Начнем. Где в самом Oracle RDBMS мы можем найти строку V\$VERSION?

Для win32-версии, эта строка имеется в файле *oracle.exe*, это легко увидеть.

Но мы также можем использовать объектные (.o) файлы от версии Oracle RDBMS для Linux, потому что в них сохраняются имена функций и глобальных переменных, а в *oracle.exe* для win32 этого нет.

Итак, строка V\$VERSION имеется в файле *kqf.o*, в самой главной Oracle-библиотеке *libserver11.a*.

Ссылка на эту текстовую строку имеется в таблице *kqfviv*, размещенной в этом же файле *kqf.o*:

#### Листинг 8.10: kqf.o

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

.rodatab:0800C4A0 kqfviw dd 0Bh ; DATA XREF: kqfchk:loc_8003A6D
.rodatab:0800C4A0 ; kqfgbn+34
.rodatab:0800C4A4 dd offset _2__STRING_10102_0 ; "GV$WAITSTAT"
.rodatab:0800C4A8 dd 4
.rodatab:0800C4AC dd offset _2__STRING_10103_0 ; "NULL"
.rodatab:0800C4B0 dd 3
.rodatab:0800C4B4 dd 0
.rodatab:0800C4B8 dd 195h
.rodatab:0800C4BC dd 4
.rodatab:0800C4C0 dd 0
.rodatab:0800C4C4 dd 0FFFFFFC1CBh
.rodatab:0800C4C8 dd 3
.rodatab:0800C4CC dd 0
.rodatab:0800C4D0 dd 0Ah
.rodatab:0800C4D4 dd offset _2__STRING_10104_0 ; "V$WAITSTAT"
.rodatab:0800C4D8 dd 4
.rodatab:0800C4DC dd offset _2__STRING_10103_0 ; "NULL"
.rodatab:0800C4E0 dd 3
.rodatab:0800C4E4 dd 0
.rodatab:0800C4E8 dd 4Eh
.rodatab:0800C4EC dd 3
.rodatab:0800C4F0 dd 0
.rodatab:0800C4F4 dd 0FFFFFFC003h
.rodatab:0800C4F8 dd 4
.rodatab:0800C4FC dd 0
.rodatab:0800C500 dd 5
.rodatab:0800C504 dd offset _2__STRING_10105_0 ; "GV$BH"
.rodatab:0800C508 dd 4
.rodatab:0800C50C dd offset _2__STRING_10103_0 ; "NULL"
.rodatab:0800C510 dd 3
.rodatab:0800C514 dd 0
.rodatab:0800C518 dd 269h
.rodatab:0800C51C dd 15h
.rodatab:0800C520 dd 0
.rodatab:0800C524 dd 0FFFFFFC1EDh
.rodatab:0800C528 dd 8
.rodatab:0800C52C dd 0
.rodatab:0800C530 dd 4
.rodatab:0800C534 dd offset _2__STRING_10106_0 ; "V$BH"
.rodatab:0800C538 dd 4
.rodatab:0800C53C dd offset _2__STRING_10103_0 ; "NULL"
.rodatab:0800C540 dd 3
.rodatab:0800C544 dd 0
.rodatab:0800C548 dd 0F5h
.rodatab:0800C54C dd 14h
.rodatab:0800C550 dd 0
.rodatab:0800C554 dd 0FFFFFFC1EEh
.rodatab:0800C558 dd 5
.rodatab:0800C55C dd 0

```

Кстати, нередко, при изучении внутренностей Oracle RDBMS, появляется вопрос, почему имена функций и глобальных переменных такие странные. Вероятно, дело в том, что Oracle RDBMS очень старый продукт сам по себе и писался

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

на Си еще в 1980-х.

А в те времена стандарт Си гарантировал поддержку имен переменных длиной только до шести символов включительно: «6 significant initial characters in an external identifier»<sup>37</sup>

Вероятно, таблица `kqfviv` содержащая в себе многие (а может даже и все) `view` с префиксом `V$`, это служебные `view` (`fixed views`), присутствующие всегда. Бегло оценив цикличность данных, мы легко видим, что в каждом элементе таблицы `kqfviv` 12 32-битных полей. В **IDA!** легко создать структуру из 12-и элементов и применить её ко всем элементам таблицы. Для версии Oracle RDBMS 11.2, здесь 1023 элемента в таблице, то есть, здесь описываются 1023 всех возможных `fixed view`. Позже, мы еще вернемся к этому числу.

Как видно, мы не очень много можем узнать чисел в этих полях. Самое первое поле всегда равно длине строки-названия `view` (без терминирующего поля).

Это справедливо для всех элементов. Но эта информация не очень полезна.

Мы также знаем, что информацию обо всех `fixed views` можно получить из `fixed view` под названием `V$FIXED_VIEW_DEFINITION` (кстати, информация для этого `view` также берется из таблиц `kqfviv` и `kqfvip`). Между прочим, там тоже 1023 элемента. Совпадение? Нет.

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='V$VERSION';

VIEW_NAME
-----
VIEW_DEFINITION
-----

V$VERSION
select  BANNER from GV$VERSION where inst_id = USERENV('Instance')
```

Итак, `V$VERSION` это как бы *thunk view* для другого, с названием `GV$VERSION`, который, в свою очередь:

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='GV$VERSION';

VIEW_NAME
-----
VIEW_DEFINITION
-----

GV$VERSION
select  inst_id, banner from x$version
```

Таблицы с префиксом `X$` в Oracle RDBMS— это также служебные таблицы, они не документированы, не могут изменяться пользователем, и обновляются динамически.

Попробуем поискать текст

<sup>37</sup>[Draft ANSI C Standard \(ANSI X3J11/88-090\) \(May 13, 1988\) \(yurichev.com\)](http://Draft ANSI C Standard (ANSI X3J11/88-090) (May 13, 1988) (yurichev.com))

```
select BANNER from GV$VERSION where inst_id =
USERENV('Instance')
```

... в файле kqf.o и находим ссылку на него в таблице kqfvip:

Листинг 8.11: kqf.o

```
.rodata:080185A0 kqfvip dd offset _2__STRING_11126_0 ; DATA XREF: kqfgvcn+18
.rodata:080185A0 ; kqfgvt+F
.rodata:080185A0 ;
"select inst_id,decode(indx,1,'data bloc"...
.rodata:080185A4 dd offset kqfv459_c_0
.rodata:080185A8 dd 0
.rodata:080185AC dd 0
...
.rodata:08019570 dd offset _2__STRING_11378_0 ;
"select BANNER from GV$VERSION where in"...
.rodata:08019574 dd offset kqfv133_c_0
.rodata:08019578 dd 0
.rodata:0801957C dd 0
.rodata:08019580 dd offset _2__STRING_11379_0 ;
"select inst_id,decode(bitand(cfflg,1),0)"...
.rodata:08019584 dd offset kqfv403_c_0
.rodata:08019588 dd 0
.rodata:0801958C dd 0
.rodata:08019590 dd offset _2__STRING_11380_0 ;
"select STATUS , NAME, IS_RECOVERY DEST"...
.rodata:08019594 dd offset kqfv199_c_0
```

Таблица, по всей видимости, имеет 4 поля в каждом элементе. Кстати, здесь так же 1023 элемента, уже знакомое нам число.

Второе поле указывает на другую таблицу, содержащую поля этого *fixed view*.

Для V\$VERSION, эта таблица только из двух элементов, первый это 6 и второй это строка BANNER (число 6 это длина строки) и далее *терминирующий* элемент содержащий 0 и нулевую Си-строку:

Листинг 8.12: kqf.o

```
.rodata:080BBAC4 kqfv133_c_0 dd 6 ; DATA XREF: .rodata:08019574
.rodata:080BBAC8 dd offset _2__STRING_5017_0 ; "BANNER"
.rodata:080BBACC dd 0
.rodata:080BBAD0 dd offset _2__STRING_0_0
```

Объединив данные из таблиц kqfviv и kqfvip, мы получим SQL-запросы, которые исполняются, когда пользователь хочет получить информацию из какого-либо *fixed view*.

Напишем программу oracle tables<sup>38</sup>, которая собирает всю эту информацию из объектных файлов от Oracle RDBMS под Linux.

<sup>38</sup>[yurichev.com](http://yurichev.com)

Для V\$VERSION, мы можем найти следующее:

Листинг 8.13: Результат работы oracle tables

```
kqfviw_element.viewname: [V$VERSION] ?: 0x3 0x43 0x1 0xffffc085 0x4
kqfvip_element.statement: [select BANNER from GV$VERSION where inst_id =
↳ USERENV('Instance')]
kqfvip_element.params:
[BANNER]
```

И:

Листинг 8.14: Результат работы oracle tables

```
kqfviw_element.viewname: [GV$VERSION] ?: 0x3 0x26 0x2 0xffffc192 0x1
kqfvip_element.statement: [select inst_id, banner from x$version]
kqfvip_element.params:
[INST_ID] [BANNER]
```

*Fixed view* GV\$VERSION отличается от V\$VERSION тем, что содержит еще и поле отражающее идентификатор *instance*.

Но так или иначе, мы теперь упираемся в таблицу X\$VERSION. Как и прочие X\$-таблицы, она не документирована, однако, мы можем оттуда что-то прочитать:

```
SQL> select * from x$version;
```

```
ADDR          INDX    INST_ID
-----
BANNER
-----
0DBAF574      0        1
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
...
```

Эта таблица содержит дополнительные поля вроде ADDR и INDX.

Бегло листая содержимое файла kqf.o в **IDA!** мы можем увидеть еще одну таблицу где есть ссылка на строку X\$VERSION, это kqftab:

Листинг 8.15: kqf.o

```
.rodata:0803CAC0      dd 9 ; element number 0x1f6
.rodata:0803CAC4      dd offset _2_STRING_13113_0 ; "X$VERSION"
.rodata:0803CAC8      dd 4
.rodata:0803CACC      dd offset _2_STRING_13114_0 ; "kqvt"
.rodata:0803CAD0      dd 4
.rodata:0803CAD4      dd 4
.rodata:0803CAD8      dd 0
.rodata:0803CADC      dd 4
.rodata:0803CAE0      dd 0Ch
.rodata:0803CAE4      dd 0FFFFFFC075h
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
.rodata:0803CAE8      dd 3
.rodata:0803CAEC      dd 0
.rodata:0803CAF0      dd 7
.rodata:0803CAF4      dd offset _2__STRING_13115_0 ; "X$KQFSZ"
.rodata:0803CAF8      dd 5
.rodata:0803CAFC      dd offset _2__STRING_13116_0 ; "kqfsz"
.rodata:0803CB00      dd 1
.rodata:0803CB04      dd 38h
.rodata:0803CB08      dd 0
.rodata:0803CB0C      dd 7
.rodata:0803CB10      dd 0
.rodata:0803CB14      dd 0FFFFFFC09Dh
.rodata:0803CB18      dd 2
.rodata:0803CB1C      dd 0
```

Здесь очень много ссылок на названия X\$-таблиц, вероятно, на все те что имеются в Oracle RDBMS этой версии.

Но мы снова упираемся в то что не имеем достаточно информации. Не ясно, что означает строка kqvt.

Вообще, префикс kq может означать *kernel* и *query*.

v, может быть, *version*, а t — *type*?

Сказать трудно.

Таблицу с очень похожим названием мы можем найти в kqf.o:

Листинг 8.16: kqf.o

```
.rodata:0808C360 kqvt_c_0 kqftap_param <4, offset _2__STRING_19_0, 917h, 0, ↵
↳ 0, 0, 4, 0, 0>
.rodata:0808C360                                ; DATA XREF:
.rodata:08042680                                ; "ADDR"
.rodata:0808C384      kqftap_param <4, offset _2__STRING_20_0, 0B02h, ↵
↳ 0, 0, 0, 4, 0, 0> ;
"INDX"
.rodata:0808C3A8      kqftap_param <7, offset _2__STRING_21_0, 0B02h, ↵
↳ 0, 0, 0, 4, 0, 0> ;
"INST_ID"
.rodata:0808C3CC      kqftap_param <6, offset _2__STRING_5017_0, 601h, ↵
↳ 0, 0, 0, 50h, 0, 0> ;
"BANNER"
.rodata:0808C3F0      kqftap_param <0, offset _2__STRING_0_0, 0, 0, 0, ↵
↳ 0, 0, 0, 0>
```

Она содержит информацию об именах полей в таблице X\$VERSION. Единственная ссылка на эту таблицу имеется в таблице kqftap:

Листинг 8.17: kqf.o

```
.rodata:08042680                                kqftap_element <0, offset kqvt_c_0, offset ↵
↳ kqvrow, 0> ; element 0x1f6
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Интересно что здесь этот элемент проходит так же под номером 0x1f6 (502-й), как и ссылка на строку X\$VERSION в таблице kqftab.

Вероятно, таблицы kqftap и kqftab дополняют друг друга, как и kqfvip и kqfviv.

Мы также видим здесь ссылку на функцию с названием kqvrow(). А вот это уже кое-что!

Сделаем так чтобы наша программа oracle tables<sup>39</sup> могла дампить и эти таблицы. Для X\$VERSION получается:

#### Листинг 8.18: Результат работы oracle tables

```
kqftab_element.name: [X$VERSION] ?: [kqvt] 0x4 0x4 0x4 0xc 0xffffc075 0x3
kqftap_param.name=[ADDR] ?: 0x917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[BANNER] ?: 0x601 0x0 0x0 0x0 0x50 0x0 0x0
kqftap_element.fn1=kqvrow
kqftap_element.fn2=NULL
```

При помощи [tracer](#), можно легко проверить, что эта функция вызывается 6 раз кряду (из функции qerfxFetch()) при получении строк из X\$VERSION.

Запустим [tracer](#) в режиме cc (он добавит комментарий к каждой исполненной инструкции):

```
tracer -a:oracle.exe bpf=oracle.exe!_kqvrow,trace:cc
```

```
_kqvrow_ proc near
var_7C = byte ptr -7Ch
var_18 = dword ptr -18h
var_14 = dword ptr -14h
Dest = dword ptr -10h
var_C = dword ptr -0Ch
var_8 = dword ptr -8
var_4 = dword ptr -4
arg_8 = dword ptr 10h
arg_C = dword ptr 14h
arg_14 = dword ptr 1Ch
arg_18 = dword ptr 20h

; FUNCTION CHUNK AT .text1:056C11A0 SIZE 00000049 BYTES

push ebp
mov ebp, esp
sub esp, 7Ch
mov eax, [ebp+arg_14] ; [EBP+1Ch]=1
mov ecx, TlsIndex ; [69AEB08h]=0
mov edx, large fs:2Ch
mov edx, [edx+ecx*4] ; [EDX+ECX*4]=0xc98c938
```

<sup>39</sup>[yurichev.com](http://yurichev.com)



```

    cmp     eax, 2          ; EAX=1
    mov     eax, [ebp+arg_8] ; [EBP+10h]=0xcdfe554
    jz      loc_2CE1288
    mov     ecx, [eax]      ; [EAX]=0..5
    mov     [ebp+var_4], edi ; EDI=0xc98c938

loc_2CE10F6: ; CODE XREF: _kqvrow_+10A
             ; _kqvrow_+1A9
    cmp     ecx, 5          ; ECX=0..5
    ja      loc_56C11C7
    mov     edi, [ebp+arg_18] ; [EBP+20h]=0
    mov     [ebp+var_14], edx ; EDX=0xc98c938
    mov     [ebp+var_8], ebx ; EBX=0
    mov     ebx, eax        ; EAX=0xcdfe554
    mov     [ebp+var_C], esi ; ESI=0xcdfe248

loc_2CE110D: ; CODE XREF: _kqvrow_+29E00E6
    mov     edx, ds:off_628B09C[ecx*4] ; [ECX*4+628B09Ch]=0x2ce1116,
    0x2ce11ac, 0x2ce11db, 0x2ce11f6, 0x2ce1236, 0x2ce127a
    jmp     edx             ; EDX=0x2ce1116, 0x2ce11ac, 0x2ce11db,
    0x2ce11f6, 0x2ce1236, 0x2ce127a

loc_2CE1116: ; DATA XREF: .rdata:off_628B09C
    push    offset aXKqvvsnBuffer ; "x$kqvvsn buffer"
    mov     ecx, [ebp+arg_C] ; [EBP+14h]=0x8a172b4
    xor     edx, edx
    mov     esi, [ebp+var_14] ; [EBP-14h]=0xc98c938
    push    edx             ; EDX=0
    push    edx             ; EDX=0
    push    50h
    push    ecx             ; ECX=0x8a172b4
    push    dword ptr [esi+10494h] ; [ESI+10494h]=0xc98cd58
    call    _kghalf        ; tracing nested maximum level (1) reached,
    skipping this CALL
    mov     esi, ds:__imp_vsnum ; [59771A8h]=0x61bc49e0
    mov     [ebp+Dest], eax ; EAX=0xce2ffb0
    mov     [ebx+8], eax    ; EAX=0xce2ffb0
    mov     [ebx+4], eax    ; EAX=0xce2ffb0
    mov     edi, [esi]      ; [ESI]=0xb200100
    mov     esi, ds:__imp_vsustr ; [597D6D4h]=0x65852148, "-
    Production"
    push    esi             ; ESI=0x65852148, "- Production"
    mov     ebx, edi        ; EDI=0xb200100
    shr     ebx, 18h        ; EBX=0xb200100
    mov     ecx, edi        ; EDI=0xb200100
    shr     ecx, 14h        ; ECX=0xb200100
    and     ecx, 0Fh        ; ECX=0xb2
    mov     edx, edi        ; EDI=0xb200100
    shr     edx, 0Ch        ; EDX=0xb200100
    movzx   edx, dl         ; DL=0
    mov     eax, edi        ; EDI=0xb200100
    shr     eax, 8          ; EAX=0xb200100
    and     eax, 0Fh        ; EAX=0xb2001
    and     edi, 0FFh       ; EDI=0xb200100

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

push    edi                ; EDI=0
mov     edi, [ebp+arg_18] ; [EBP+20h]=0
push    eax                ; EAX=1
mov     eax, ds:__imp__vsnb ;
[597D6D8h]=0x65852100, "Oracle Database 11g Enterprise Edition Release %d.%d.%d.%d %s"
push    edx                ; EDX=0
push    ecx                ; ECX=2
push    ebx                ; EBX=0xb
mov     ebx, [ebp+arg_8] ; [EBP+10h]=0xcdfe554
push    eax                ;
EAX=0x65852100, "Oracle Database 11g Enterprise Edition Release %d.%d.%d.%d %s"
mov     eax, [ebp+Dest] ; [EBP-10h]=0xce2ffb0
push    eax                ; EAX=0xce2ffb0
call    ds:__imp__sprintf ; op1=MSVCR80.dll!sprintf tracing nested
maximum level (1) reached, skipping this CALL
add     esp, 38h
mov     dword ptr [ebx], 1

loc_2CE1192: ; CODE XREF: _kqvrow_+FB
          ; _kqvrow_+128 ...
test    edi, edi          ; EDI=0
jnz     __VInfreq__kqvrow
mov     esi, [ebp+var_C] ; [EBP-0Ch]=0xcdfe248
mov     edi, [ebp+var_4] ; [EBP-4]=0xc98c938
mov     eax, ebx          ; EBX=0xcdfe554
mov     ebx, [ebp+var_8] ; [EBP-8]=0
lea     eax, [eax+4]      ; [EAX+4]=0xce2ffb0, "NLSRTL Version
11.2.0.1.0 - Production", "Oracle Database 11g Enterprise Edition Release
11.2.0.1.0 - Production", "PL/SQL Release 11.2.0.1.0 - Production", "TNS
for 32-bit Windows: Version 11.2.0.1.0 - Production"

loc_2CE11A8: ; CODE XREF: _kqvrow_+29E00F6
mov     esp, ebp
pop     ebp
retn                                ; EAX=0xcdfe558

loc_2CE11AC: ; DATA XREF: .rdata:0628B0A0
mov     edx, [ebx+8]      ; [EBX+8]=0xce2ffb0, "Oracle Database 11g
Enterprise Edition Release 11.2.0.1.0 - Production"
mov     dword ptr [ebx], 2
mov     [ebx+4], edx      ; EDX=0xce2ffb0, "Oracle Database 11g
Enterprise Edition Release 11.2.0.1.0 - Production"
push    edx              ; EDX=0xce2ffb0, "Oracle Database 11g
Enterprise Edition Release 11.2.0.1.0 - Production"
call    __kxvsn          ; tracing nested maximum level (1) reached,
skipping this CALL
pop     ecx
mov     edx, [ebx+4]      ; [EBX+4]=0xce2ffb0, "PL/SQL Release
11.2.0.1.0 - Production"
movzx   ecx, byte ptr [edx] ; [EDX]=0x50
test    ecx, ecx         ; ECX=0x50
jnz     short loc_2CE1192
mov     edx, [ebp+var_14]
mov     esi, [ebp+var_C]
mov     eax, ebx
mov     ebx, [ebp+var_8]

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

        mov     ecx, [eax]
        jmp     loc_2CE10F6

loc_2CE11DB: ; DATA XREF: .rdata:0628B0A4
        push   0
        push   50h
        mov     edx, [ebx+8] ; [EBX+8]=0xce2ffb0, "PL/SQL Release
11.2.0.1.0 - Production"
        mov     [ebx+4], edx ; EDX=0xce2ffb0, "PL/SQL Release 11.2.0.1.0
- Production"
        push   edx ; EDX=0xce2ffb0, "PL/SQL Release 11.2.0.1.0
- Production"
        call   _lmxver ; tracing nested maximum level (1) reached,
skipping this CALL
        add     esp, 0Ch
        mov     dword ptr [ebx], 3
        jmp     short loc_2CE1192

loc_2CE11F6: ; DATA XREF: .rdata:0628B0A8
        mov     edx, [ebx+8] ; [EBX+8]=0xce2ffb0
        mov     [ebp+var_18], 50h
        mov     [ebx+4], edx ; EDX=0xce2ffb0
        push   0
        call   _npinli ; tracing nested maximum level (1) reached,
skipping this CALL
        pop     ecx
        test    eax, eax ; EAX=0
        jnz    loc_56C11DA
        mov     ecx, [ebp+var_14] ; [EBP-14h]=0xc98c938
        lea    edx, [ebp+var_18] ; [EBP-18h]=0x50
        push   edx ; EDX=0xd76c93c
        push   dword ptr [ebx+8] ; [EBX+8]=0xce2ffb0
        push   dword ptr [ecx+13278h] ; [ECX+13278h]=0xacce190
        call   _nrtnsvrs ; tracing nested maximum level (1) reached,
skipping this CALL
        add     esp, 0Ch

loc_2CE122B: ; CODE XREF: _kqvrow_+29E0118
        mov     dword ptr [ebx], 4
        jmp     loc_2CE1192

loc_2CE1236: ; DATA XREF: .rdata:0628B0AC
        lea    edx, [ebp+var_7C] ; [EBP-7Ch]=1
        push   edx ; EDX=0xd76c8d8
        push   0
        mov     esi, [ebx+8] ; [EBX+8]=0xce2ffb0, "TNS for 32-bit
Windows: Version 11.2.0.1.0 - Production"
        mov     [ebx+4], esi ; ESI=0xce2ffb0, "TNS for 32-bit Windows:
Version 11.2.0.1.0 - Production"
        mov     ecx, 50h
        mov     [ebp+var_18], ecx ; ECX=0x50
        push   ecx ; ECX=0x50
        push   esi ; ESI=0xce2ffb0, "TNS for 32-bit Windows:
Version 11.2.0.1.0 - Production"
        call   _lxvers ; tracing nested maximum level (1) reached,
skipping this CALL

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

    add     esp, 10h
    mov     edx, [ebp+var_18] ; [EBP-18h]=0x50
    mov     dword ptr [ebx], 5
    test    edx, edx          ; EDX=0x50
    jnz     loc_2CE1192
    mov     edx, [ebp+var_14]
    mov     esi, [ebp+var_C]
    mov     eax, ebx
    mov     ebx, [ebp+var_8]
    mov     ecx, 5
    jmp     loc_2CE10F6

loc_2CE127A: ; DATA XREF: .rdata:0628B0B0
    mov     edx, [ebp+var_14] ; [EBP-14h]=0xc98c938
    mov     esi, [ebp+var_C] ; [EBP-0Ch]=0xcdfe248
    mov     edi, [ebp+var_4] ; [EBP-4]=0xc98c938
    mov     eax, ebx          ; EBX=0xcdfe554
    mov     ebx, [ebp+var_8] ; [EBP-8]=0

loc_2CE1288: ; CODE XREF: _kqvrow_+1F
    mov     eax, [eax+8]      ; [EAX+8]=0xce2ffb0, "NLSRTL Version
11.2.0.1.0 - Production"
    test    eax, eax          ; EAX=0xce2ffb0, "NLSRTL Version 11.2.0.1.0
- Production"
    jz     short loc_2CE12A7
    push    offset aXKqvvsnBuffer ; "x$kqvvsn buffer"
    push    eax                ; EAX=0xce2ffb0, "NLSRTL Version 11.2.0.1.0
- Production"
    mov     eax, [ebp+arg_C] ; [EBP+14h]=0x8a172b4
    push    eax                ; EAX=0x8a172b4
    push    dword ptr [edx+10494h] ; [EDX+10494h]=0xc98cd58
    call    _kghfrf            ; tracing nested maximum level (1) reached,
skipping this CALL
    add     esp, 10h

loc_2CE12A7: ; CODE XREF: _kqvrow_+1C1
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    retn                                ; EAX=0
_kqvrow_  endp

```

Так можно легко увидеть, что номер строки таблицы задается извне. Сама функция возвращает строку, формируя её так:

Строка 1	Использует глобальные переменные vsnstr, vsnnum, vsnban. Вызывает sprintf().
Строка 2	Вызывает kkvvsn().
Строка 3	Вызывает lmxver().
Строка 4	Вызывает npinli(), nrtnsvrs().
Строка 5	Вызывает lxvers().

Так вызываются соответствующие функции для определения номеров версий отдельных модулей.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

### 8.11.2. Таблица X\$KSMLRU в Oracle RDBMS

В заметке *Diagnosing and Resolving Error ORA-04031 on the Shared Pool or Other Memory Pools [Video] [ID 146599.1]* упоминается некая служебная таблица:

There is a fixed table called X\$KSMLRU that tracks allocations in the shared pool that cause other objects in the shared pool to be aged out. This fixed table can be used to identify what is causing the large allocation.

If many objects are being periodically flushed from the shared pool then this will cause response time problems and will likely cause library cache latch contention problems when the objects are reloaded into the shared pool.

One unusual thing about the X\$KSMLRU fixed table is that the contents of the fixed table are erased whenever someone selects from the fixed table. This is done since the fixed table stores only the largest allocations that have occurred. The values are reset after being selected so that subsequent large allocations can be noted even if they were not quite as large as others that occurred previously. Because of this resetting, the output of selecting from this table should be carefully kept since it cannot be retrieved back after the query is issued.

Однако, как можно легко убедиться, эта системная таблица очищается всякий раз, когда кто-то делает запрос к ней. Сможем ли мы найти причину, почему это происходит? Если вернуться к уже рассмотренным таблицам kqftab и kqftap полученных при помощи oracle tables<sup>40</sup>, содержащим информацию о X\$-таблицах, мы узнаем что для того чтобы подготовить строки этой таблицы, вызывается функция ksmlrs():

Листинг 8.19: Результат работы oracle tables

```
kqftab_element.name: [X$KSMLRU] ?: [ksmlr] 0x4 0x64 0x11 0xc 0xffffc0bb 0x5
kqftap_param.name=[ADDR] ?: 0x917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRIDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRDUR] ?: 0xb02 0x0 0x0 0x0 0x4 0x4 0x0
kqftap_param.name=[KSMLRSHRPOOL] ?: 0xb02 0x0 0x0 0x0 0x4 0x8 0x0
kqftap_param.name=[KSMLRCOM] ?: 0x501 0x0 0x0 0x0 0x14 0xc 0x0
kqftap_param.name=[KSMLRSIZ] ?: 0x2 0x0 0x0 0x0 0x4 0x20 0x0
kqftap_param.name=[KSMLRNUM] ?: 0x2 0x0 0x0 0x0 0x4 0x24 0x0
kqftap_param.name=[KSMLRHON] ?: 0x501 0x0 0x0 0x0 0x20 0x28 0x0
kqftap_param.name=[KSMLROHV] ?: 0xb02 0x0 0x0 0x0 0x4 0x48 0x0
kqftap_param.name=[KSMLRSES] ?: 0x17 0x0 0x0 0x0 0x4 0x4c 0x0
kqftap_param.name=[KSMLRADU] ?: 0x2 0x0 0x0 0x0 0x4 0x50 0x0
kqftap_param.name=[KSMLRNID] ?: 0x2 0x0 0x0 0x0 0x4 0x54 0x0
kqftap_param.name=[KSMLRNSD] ?: 0x2 0x0 0x0 0x0 0x4 0x58 0x0
kqftap_param.name=[KSMLRNCD] ?: 0x2 0x0 0x0 0x0 0x4 0x5c 0x0
kqftap_param.name=[KSMLRNED] ?: 0x2 0x0 0x0 0x0 0x4 0x60 0x0
```

<sup>40</sup>[yurichev.com](http://yurichev.com)

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
kqftap_element.fn1=ksmlrs
kqftap_element.fn2=NULL
```

Действительно, при помощи [tracer](#) легко убедиться, что эта функция вызывается каждый раз, когда мы обращаемся к таблице X\$KSMLRU.

Здесь есть ссылки на функции `ksmsplu_sp()` и `ksmsplu_jp()`, каждая из которых в итоге вызывает `ksmsplu()`. В конце функции `ksmsplu()` мы видим вызов `memset()`:

Листинг 8.20: ksm.o

```
...
.text:00434C50 loc_434C50: ; DATA XREF: .rdata:off_5E50EA8
.text:00434C50 mov edx, [ebp-4]
.text:00434C53 mov [eax], esi
.text:00434C55 mov esi, [edi]
.text:00434C57 mov [eax+4], esi
.text:00434C5A mov [edi], eax
.text:00434C5C add edx, 1
.text:00434C5F mov [ebp-4], edx
.text:00434C62 jnz loc_434B7D
.text:00434C68 mov ecx, [ebp+14h]
.text:00434C6B mov ebx, [ebp-10h]
.text:00434C6E mov esi, [ebp-0Ch]
.text:00434C71 mov edi, [ebp-8]
.text:00434C74 lea eax, [ecx+8Ch]
.text:00434C7A push 370h ; Size
.text:00434C7F push 0 ; Val
.text:00434C81 push eax ; Dst
.text:00434C82 call __intel_fast_memset
.text:00434C87 add esp, 0Ch
.text:00434C8A mov esp, ebp
.text:00434C8C pop ebp
.text:00434C8D retn
.text:00434C8D _ksmsplu endp
```

Такие конструкции (`memset (block, 0, size)`) очень часто используются для простого обнуления блока памяти. Мы можем попробовать рискнуть, заблокировав вызов `memset()` и посмотреть, что будет?

Запускаем [tracer](#) со следующей опцией: поставить точку останова на `0x434C7A` (там, где начинается передача параметров для функции `memset()`) так, чтобы [tracer](#) в этом месте установил указатель инструкций процессора (EIP) на место, где уже произошла очистка переданных параметров в `memset()` (по адресу `0x434C8A`):

Можно сказать, при помощи этого, мы симулируем безусловный переход с адреса `0x434C7A` на `0x434C8A`.

```
tracer -a:oracle.exe bpx=oracle.exe!0x00434C7A,set(eip,0x00434C8A)
```

(Важно: все эти адреса справедливы только для win32-версии Oracle RDBMS 11.2)

Действительно, после этого мы можем обращаться к таблице X\$KSMLRU сколько угодно, и она уже не очищается!

На всякий случай, не повторяйте этого на своих production-серверах.

Впрочем, это не обязательно полезное или желаемое поведение системы, но как эксперимент по поиску нужного кода, нам это подошло!

### 8.11.3. Таблица V\$TIMER в Oracle RDBMS

V\$TIMER это еще один служебный *fixed view*, отражающий какое-то часто меняющееся значение:

V\$TIMER displays the elapsed time in hundredths of a second. Time is measured since the beginning of the epoch, which is operating system specific, and wraps around to 0 again whenever the value overflows four bytes (roughly 497 days).

(Из документации Oracle RDBMS<sup>41</sup>)

Интересно что периоды разные в Oracle для Win32 и для Linux. Сможем ли мы найти функцию, отвечающую за генерирование этого значения?

Как видно, эта информация, в итоге, берется из системной таблицы X\$KSUTM.

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='V$TIMER';

VIEW_NAME
-----
VIEW_DEFINITION
-----

V$TIMER
select  HSECS from GV$TIMER where inst_id = USERENV('Instance')

SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='GV$TIMER';

VIEW_NAME
-----
VIEW_DEFINITION
-----

GV$TIMER
select inst_id,ksutmtim from x$ksutm
```

Здесь мы упираемся в небольшую проблему, в таблицах kqftab/kqftar нет указателей на функцию, которая бы генерировала значение:

<sup>41</sup>[http://docs.oracle.com/cd/B28359\\_01/server.111/b28320/dynviews\\_3104.htm](http://docs.oracle.com/cd/B28359_01/server.111/b28320/dynviews_3104.htm)

## Листинг 8.21: Результат работы oracle tables

```
kqftab_element.name: [X$KSUTM] ?: [ksutm] 0x1 0x4 0x4 0x0 0xffffc09b 0x3
kqftap_param.name=[ADDR] ?: 0x10917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0x20b02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSUTMTIM] ?: 0x1302 0x0 0x0 0x0 0x4 0x0 0x1e
kqftap_element.fn1=NULL
kqftap_element.fn2=NULL
```

Попробуем в таком случае просто поискать строку KSUTMTIM, и находим ссылку на нее в такой функции:

```
kqfd_DRN_ksutm_c proc near      ; DATA XREF: .rodata:0805B4E8

arg_0    = dword ptr  8
arg_8    = dword ptr  10h
arg_C    = dword ptr  14h

        push    ebp
        mov     ebp, esp
        push   [ebp+arg_C]
        push   offset ksugtm
        push   offset _2_STRING_1263_0 ; "KSUTMTIM"
        push   [ebp+arg_8]
        push   [ebp+arg_0]
        call   kqfd_cfui_drain
        add    esp, 14h
        mov    esp, ebp
        pop    ebp
        retn
kqfd_DRN_ksutm_c endp
```

Сама функция kqfd\_DRN\_ksutm\_c() упоминается в таблице kqfd\_tab\_registry\_0 вот так:

```
dd offset _2_STRING_62_0 ; "X$KSUTM"
dd offset kqfd_OPN_ksutm_c
dd offset kqfd_tabl_fetch
dd 0
dd 0
dd offset kqfd_DRN_ksutm_c
```

Упоминается также некая функция ksugtm(). Посмотрим, что там (в Linux x86):

## Листинг 8.22: ksu.o

```
ksugtm proc near

var_1C   = byte ptr -1Ch
arg_4    = dword ptr  0Ch

        push    ebp
        mov     ebp, esp
```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



```

sub     esp, 1Ch
lea     eax, [ebp+var_1C]
push   eax
call   slgcs
pop     ecx
mov     edx, [ebp+arg_4]
mov     [edx], eax
mov     eax, 4
mov     esp, ebp
pop     ebp
retn
ksugtm endp

```

В win32-версии тоже самое.

Искомая ли эта функция? Попробуем узнать:

```
tracer -a:oracle.exe bpf=oracle.exe!_ksugtm,args:2,dump_args:0x4
```

Пробуем несколько раз:

```

SQL> select * from V$TIMER;

      HSECS
-----
27294929

SQL> select * from V$TIMER;

      HSECS
-----
27295006

SQL> select * from V$TIMER;

      HSECS
-----
27295167

```

Листинг 8.23: вывод `tracer`

```

TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!↵
↳ __VInfreq__qerfxFetch+0xfad (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                                "8.                                ↵
↳ "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: D1 7C A0 01                                ".|..                               ↵
↳ "
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!↵
↳ __VInfreq__qerfxFetch+0xfad (0x56bb6d5))
Argument 2/2

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

0D76C5F0: 38 C9          "8.          ↗
  ↘ "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: 1E 7D A0 01   ".}..       ↗
  ↘ "
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!
  ↘ __VInfreq__qerfxFetch+0xfad (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9          "8.          ↗
  ↘ "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: BF 7D A0 01   ".}..       ↗
  ↘ "

```

Действительно — значение то, что мы видим в SQL\*Plus, и оно возвращается через второй аргумент.

Посмотрим, что в функции `slgcs()` (Linux x86):

```

slgcs  proc near
var_4  = dword ptr -4
arg_0  = dword ptr  8

    push    ebp
    mov     ebp, esp
    push    esi
    mov     [ebp+var_4], ebx
    mov     eax, [ebp+arg_0]
    call   $+5
    pop     ebx
    nop
    ; PIC mode
    mov     ebx, offset _GLOBAL_OFFSET_TABLE_
    mov     dword ptr [eax], 0
    call   sltrgtime64 ; PIC mode
    push    0
    push    0Ah
    push    edx
    push    eax
    call   __udivdi3 ; PIC mode
    mov     ebx, [ebp+var_4]
    add     esp, 10h
    mov     esp, ebp
    pop     ebp
    retn
slgcs  endp

```

(это просто вызов `sltrgtime64()` и деление его результата на 10 (?? (стр. ??)))

И в win32-версии:

```

_slgcs  proc near ; CODE XREF: _dbgfgHtElResetCount+15

```

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; _dbgcrRunActions+1528
db      66h
nop
push    ebp
mov     ebp, esp
mov     eax, [ebp+8]
mov     dword ptr [eax], 0
call    ds:__imp_GetTickCount@0 ; GetTickCount()
mov     edx, eax
mov     eax, 0CCCCCDh
mul     edx
shr     edx, 3
mov     eax, edx
mov     esp, ebp
pop     ebp
retn
_slgcs  endp

```

Это просто результат `GetTickCount()` <sup>42</sup> поделенный на 10 (?? (стр. ??)).

Уаля! Вот почему в win32-версии и версии Linux x86 разные результаты, потому что они получаются разными системными функциями **OS!**

*Drain* по-английски *дренаж, отток, водосток*. Таким образом, возможно имеется ввиду *подключение* определенного столбца системной таблице к функции.

Добавим поддержку таблицы `kqfd_tab_registry_0` в *oracle tables*<sup>43</sup>, теперь мы можем видеть, при помощи каких функций, столбцы в системных таблицах *подключаются* к значениям, например:

```

[X$KSUTM] [kqfd_OPN_ksutm_c] [kqfd_tabl_fetch] [NULL] [NULL] [↵
↳ kqfd_DRN_ksutm_c]
[X$KSUSGIF] [kqfd_OPN_ksusg_c] [kqfd_tabl_fetch] [NULL] [NULL] [↵
↳ kqfd_DRN_ksusg_c]

```

*OPN*, возможно, *open*, а *DRN*, вероятно, означает *drain*.

## 8.12. Вручную написанный на ассемблере код

### 8.12.1. Тестовый файл EICAR

Этот .COM-файл предназначен для тестирования антивирусов, его можно запустить в MS-DOS и он выведет такую строку: «EICAR-STANDARD-ANTIVIRUS-TEST-FILE!».

Он примечателен тем, что он полностью состоит только из печатных ASCII-символов, следовательно, его можно набрать в любом текстовом редакторе:

```
X50!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

<sup>42</sup>MSDN

<sup>43</sup>yurichev.com

Попробуем его разобрать:

```

; изначальное состояние: SP=0FFFEh, SS:[SP]=0
0100 58          pop      ax
; AX=0, SP=0
0101 35 4F 21    xor      ax, 214Fh
; AX = 214Fh and SP = 0
0104 50          push     ax
; AX = 214Fh, SP = FFFEh and SS:[FFFE] = 214Fh
0105 25 40 41    and      ax, 4140h
; AX = 140h, SP = FFFEh and SS:[FFFE] = 214Fh
0108 50          push     ax
; AX = 140h, SP = FFFCh, SS:[FFFC] = 140h and SS:[FFFE] = 214Fh
0109 5B          pop      bx
; AX = 140h, BX = 140h, SP = FFFEh and SS:[FFFE] = 214Fh
010A 34 5C       xor      al, 5Ch
; AX = 11Ch, BX = 140h, SP = FFFEh and SS:[FFFE] = 214Fh
010C 50          push     ax
010D 5A          pop      dx
; AX = 11Ch, BX = 140h, DX = 11Ch, SP = FFFEh and SS:[FFFE] = 214Fh
010E 58          pop      ax
; AX = 214Fh, BX = 140h, DX = 11Ch and SP = 0
010F 35 34 28    xor      ax, 2834h
; AX = 97Bh, BX = 140h, DX = 11Ch and SP = 0
0112 50          push     ax
0113 5E          pop      si
; AX = 97Bh, BX = 140h, DX = 11Ch, SI = 97Bh and SP = 0
0114 29 37       sub      [bx], si
0116 43          inc      bx
0117 43          inc      bx
0118 29 37       sub      [bx], si
011A 7D 24       jge     short near ptr word_10140
011C 45 49 43 ... db 'EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$'
0140 48 2B      word_10140 dw 2B48h ; CD 21 (INT 21) будет здесь
0142 48 2A      dw 2A48h ; CD 20 (INT 20) будет здесь
0144 0D          db      0Dh
0145 0A          db      0Ah

```

Добавим везде комментарии, показывающие состояние регистров и стека после каждой инструкции.

Собственно, все эти инструкции нужны только для того чтобы исполнить следующий код:

```

B4 09      MOV AH, 9
BA 1C 01   MOV DX, 11Ch
CD 21      INT 21h
CD 20      INT 20h

```

INT 21h с функцией 9 (переданной в AH) просто выводит строку, адрес которой передан в DS:DX. Кстати, строка должна быть завершена символом '\$'. Надо полагать, это наследие CP/M и эта функция в DOS осталась для совместимости. INT 20h возвращает управление в DOS.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Но, как видно, далеко не все опкоды этих инструкций печатные. Так что основная часть EICAR-файла это:

- подготовка нужных значений регистров (AH и DX);
- подготовка в памяти опкодов для INT 21 и INT 20;
- исполнение INT 21 и INT 20.

Кстати, подобная техника широко используется для создания шелл-кодов, где нужно создать x86-код, который будет нужно передать в виде текстовой строки.

Здесь также список всех x86-инструкций с печатаемыми опкодами: ?? (стр. ??).

## 8.13. Демо

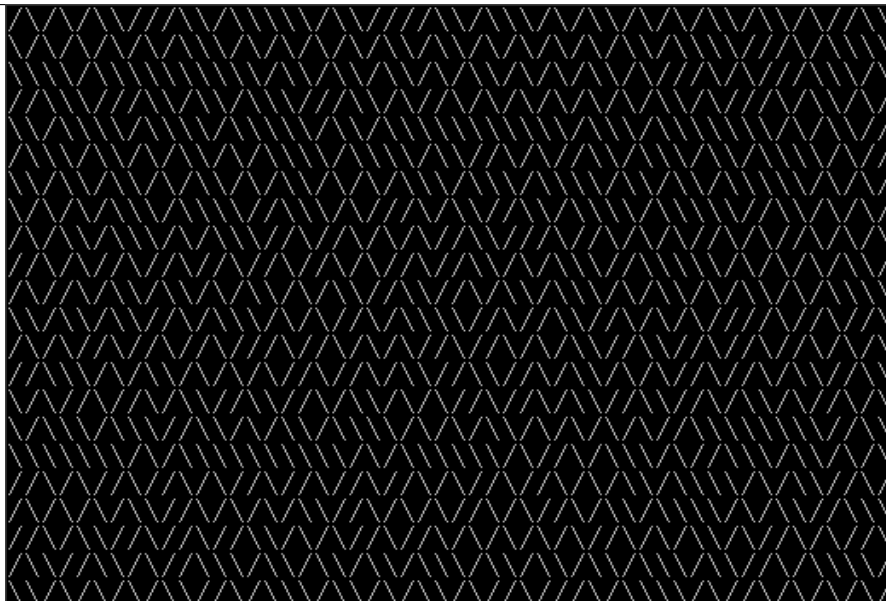
Демо (или демомейкинг) были великолепным упражнением в математике, программировании компьютерной графики и очень плотному программированию на ассемблере вручную.

### 8.13.1. 10 PRINT CHR\$(205.5+RND(1)); : GOTO 10

Все примеры здесь для .COM-файлов под MS-DOS.

В [Nick Montfort et al, *10 PRINT CHR\$(205.5+RND(1)); : GOTO 10*, (The MIT Press:2012)]<sup>44</sup> можно прочитать об одном из простейших генераторов случайных лабиринтов. Он просто бесконечно и случайно печатает символ слэша или обратный слэш, выдавая в итоге что-то вроде:

<sup>44</sup>Также доступно здесь: [http://trope-tank.mit.edu/10\\_PRINT\\_121114.pdf](http://trope-tank.mit.edu/10_PRINT_121114.pdf)



Здесь несколько известных реализаций для 16-битного x86.

### Версия 42-х байт от Trixter

Листинг взят с его сайта<sup>45</sup>, но комментарии — автора.

```

00000000: B001      mov     al,1      ; установить видеорежим 40x25
00000002: CD10      int     010      ; установить видеостраницу для
00000004: 30FF      xor     bh,bh     ; установка видеостраницы для
; вызова int 10h
00000006: B9D007    mov     cx,007D0 ; вывод 2000 символов
00000009: 31C0      xor     ax,ax
0000000B: 9C        pushf             ; сохранить флаги
; узнать случайное число из чипа таймера
0000000C: FA        cli             ; запретить прерывания
0000000D: E643      out     043,al   ; записать 0 в порт 43h
; прочитать 16-битное значение из порта 40h
0000000F: E440      in     al,040
00000011: 88C4      mov     ah,al
00000013: E440      in     al,040
00000015: 9D        popf             ; разрешить прерывания
; возвращая значение флага IF
00000016: 86C4      xchg    ah,al
; здесь мы имеем псевдослучайное 16-битное значение
00000018: D1E8      shr     ax,1
0000001A: D1E8      shr     ax,1
; в CF сейчас находится второй бит из значения
0000001C: B05C      mov     al,05C ; '
; если CF=1, пропускаем следующую инструкцию
0000001E: 7202      jc     00000022

```

<sup>45</sup><http://trixter.oldskool.org/2012/12/17/maze-generation-in-thirteen-bytes/>

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

; если CF=0, загружаем в регистр AL другой символ
00000020: B02F      mov     al,02F ; '/'
; вывод символа
00000022: B40E      mov     ah,00E
00000024: CD10      int     010
00000026: E2E1      loop   00000009 ; цикл в 2000 раз
00000028: CD20      int     020      ; возврат в DOS

```

Псевдослучайное число на самом деле это время, прошедшее со старта системы, получаемое из чипа таймера 8253, это значение увеличивается на единицу 18.2 раза в секунду.

Записывая ноль в порт 43h, мы имеем ввиду что команда это «выбрать счетчик 0», "counter latch", "двоичный счетчик" (а не значение **BCD!**).

Прерывания снова разрешаются при помощи инструкции POPF, которая также возвращает флаг IF.

Инструкцию IN нельзя использовать с другими регистрами кроме AL, поэтому здесь перетасовка.

### Моя попытка укоротить версию Trixter: 27 байт

Мы можем сказать, что мы используем таймер не для того чтобы получить точное время, но псевдослучайное число, так что мы можем не тратить время (и код) на запрещение прерываний. Еще можно сказать, что так как мы берем бит из младшей 8-битной части, то мы можем считывать только её.

Немного укоротим код и выходит 27 байт:

```

00000000: B9D007  mov     cx,007D0 ; вывести только 2000 символов
00000003: 31C0     xor     ax,ax     ; команда чипу таймера
00000005: E643     out     043,al
00000007: E440     in     al,040    ; читать 8 бит из таймера
00000009: D1E8     shr     ax,1     ; переместить второй бит в флаг CF
0000000B: D1E8     shr     ax,1
0000000D: B05C     mov     al,05C   ; подготовить '\'
0000000F: 7202     jc     00000013
00000011: B02F     mov     al,02F   ; подготовить '/'
; вывести символ на экран
00000013: B40E     mov     ah,00E
00000015: CD10     int     010
00000017: E2EA     loop   00000003
; выход в DOS
00000019: CD20     int     020

```

### Использование случайного мусора в памяти как источника случайных чисел

Так как это MS-DOS, защиты памяти здесь нет вовсе, так что мы можем читать с какого угодно адреса. И даже более того: простая инструкция LODSB будет читать байт по адресу DS:SI, но это не проблема если правильные значения

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

не установлены в регистры, пусть она читает 1) случайные байты; 2) из случайного места в памяти!

Так что на странице Trixter-а<sup>46</sup> можно найти предложение использовать LODSB без всякой инициализации.

Есть также предложение использовать инструкцию SCASB вместо, потому что она выставляет флаги в соответствии с прочитанным значением.

Еще одна идея насчет минимизации кода — это использовать прерывание DOS INT 29h которое просто печатает символ на экране из регистра AL.

Это то что сделал Peter Ferrie<sup>47</sup>:

Листинг 8.24: Peter Ferrie: 10 байт

```

; AL в этом месте имеет случайное значение
00000000: AE      scasb
; CF устанавливается по результату вычитания случайного байта памяти из AL.
; так что он здесь случаен, в каком-то смысле
00000001: D6      setalc
; AL выставляется в 0xFF если CF=1 или в 0 если наоборот
00000002: 242D   and     al,02D ; '-'
; AL здесь 0x2D либо 0
00000004: 042F   add     al,02F ; '/'
; AL здесь 0x5C либо 0x2F
00000006: CD29   int     029      ; вывести AL на экране
00000008: EBF6   jmps   00000000 ; бесконечный цикл

```

Так что можно избавиться и от условных переходов. **ASCII!**-код обратного слэша («\») это 0x5C и 0x2F для слэша («/»).

Так что нам нужно конвертировать один (псевдослучайный) бит из флага CF в значение 0x5C или 0x2F.

Это делается легко: применяя операцию «И» ко всем битам в AL (где все 8 бит либо выставлены, либо сброшены) с 0x2D мы имеем просто 0 или 0x2D.

Прибавляя значение 0x2F к этому значению, мы получаем 0x5C или 0x2F. И просто выводим это на экран.

## Вывод

Также стоит отметить, что результат может быть разным в эмуляторе DOSBox, Windows NT и даже MS-DOS, из-за разных условий: чип таймера может эмулироваться по-разному, изначальные значения регистров также могут быть разными.

<sup>46</sup><http://trixter.oldschool.org/2012/12/17/maze-generation-in-thirteen-bytes/>

<sup>47</sup><http://pferrie.host22.com/misc/10print.htm>

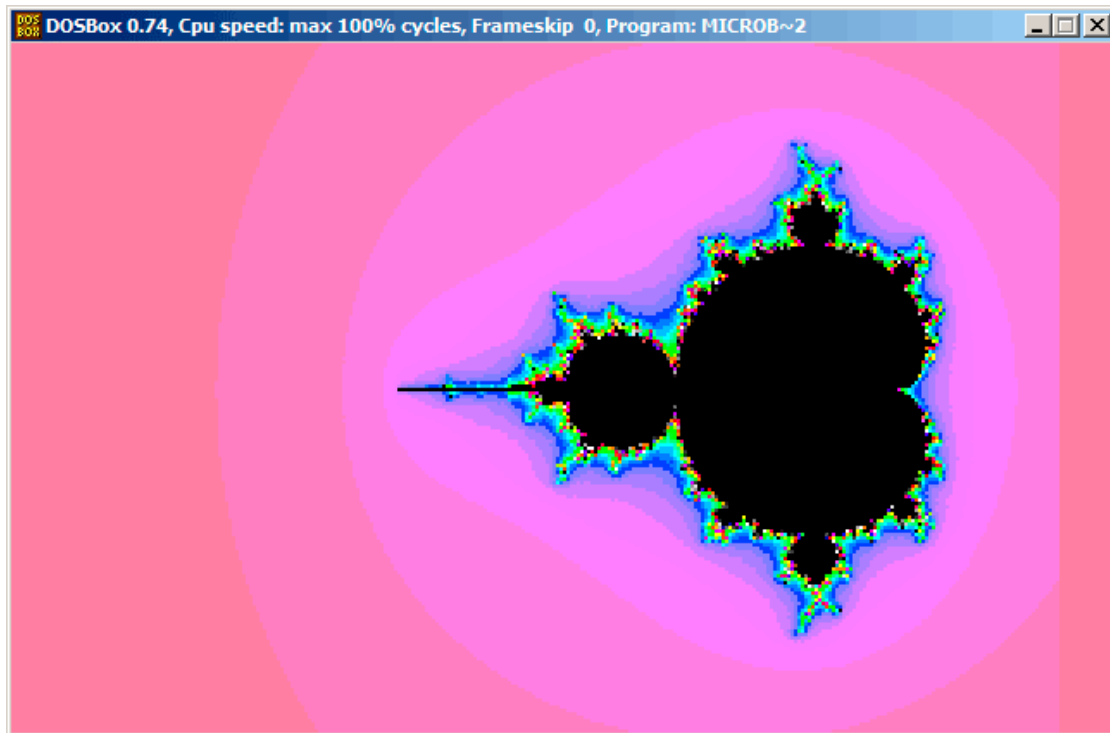


### 8.13.2. Множество Мандельброта

В Mathematical Recipes<sup>48</sup> есть некоторые важные замечания о теории вокруг множества Мандельброта.

Вот демо<sup>49</sup> написанное автором по имени «Sir\_Lagsalot» в 2009, рисующее множество Мандельброта, и это программа для x86 с размером файла всего 64 байта. Там только 30 16-битных x86-инструкций.

Вот что она рисует:



Попробуем разобраться, как она работает.

Демо, хотя и крошечная (только 64 байта или 30 инструкций), реализует общий алгоритм, изложенный здесь, но с некоторыми трюками.

Исходный код можно скачать, так что вот он, но также снабдим его своими комментариями:

Листинг 8.25: Исходный код с комментариями

```

1 ; X это столбец на экране
2 ; Y это строка на экране
3
4
5 ; X=0, Y=0 X=319, Y=0

```

<sup>48</sup><https://math.recipes>

<sup>49</sup>Можно скачать [здесь](#),

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

6 ; +----->
7 ; |
8 ; |
9 ; |
10 ; |
11 ; |
12 ; |
13 ; v
14 ; X=0, Y=199 X=319, Y=199
15
16
17 ; переключиться в графический видеорежим VGA 320*200*256
18 mov al,13h
19 int 10h
20 ; в самом начале BX равен 0
21 ; в самом начале DI равен 0xFFFFE
22 ; DS:BX (или DS:0) указывает на Program Segment Prefix в этот момент
23 ; ... первые 4 байта которого этого CD 20 FF 9F
24 les ax,[bx]
25 ; ES:AX=9FFF:20CD
26
27 FillLoop:
28 ; установить DX в 0. CWD работает так: DX:AX = sign_extend(AX).
29 ; AX здесь 0x20CD (в начале) или меньше 320 (когда вернемся после цикла),
30 ; так что DX всегда будет 0.
31 cwd
32 mov ax,di
33 ; AX это текущий указатель внутри VGA-буфера
34 ; разделить текущий указатель на 320
35 mov cx,320
36 div cx
37 ; DX (start_X) - остаток (столбец: 0..319); AX - результат (строка: 0..199)
38 sub ax,100
39 ; AX=AX-100, так что AX (start_Y) сейчас в пределах -100..99
40 ; DX в пределах 0..319 или 0x0000..0x013F
41 dec dh
42 ; DX сейчас в пределах 0xFF00..0x003F (-256..63)
43
44 xor bx,bx
45 xor si,si
46 ; BX (temp_X)=0; SI (temp_Y)=0
47
48 ; получить максимальное количество итераций
49 ; CX всё ещё 320 здесь, так что это будет максимальным количеством итераций
50 MandelLoop:
51 mov bp,si ; BP = temp_Y
52 imul si,bx ; SI = temp_X*temp_Y
53 add si,si ; SI = SI*2 = (temp_X*temp_Y)*2
54 imul bx,bx ; BX = BX^2 = temp_X^2
55 jo MandelBreak ; переполнение?
56 imul bp,bp ; BP = BP^2 = temp_Y^2
57 jo MandelBreak ; переполнение?
58 add bx,bp ; BX = BX+BP = temp_X^2 + temp_Y^2

```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```

59  jo MandelBreak ; переполнение?
60  sub bx,bp      ; BX = BX-BP = temp_X^2 + temp_Y^2 - temp_Y^2 = temp_X^2
61  sub bx,bp      ; BX = BX-BP = temp_X^2 - temp_Y^2
62
63  ; скорректировать масштаб:
64  sar bx,6      ; BX=BX/64
65  add bx,dx      ; BX=BX+start_X
66  ; здесь temp_X = temp_X^2 - temp_Y^2 + start_X
67  sar si,6      ; SI=SI/64
68  add si,ax      ; SI=SI+start_Y
69  ; здесь temp_Y = (temp_X*temp_Y)*2 + start_Y
70
71  loop MandelLoop
72
73  MandelBreak:
74  ; CX=итерации
75  xchg ax,cx
76  ; AX=итерации. записать AL в VGA-буфер на ES:[DI]
77  stosb
78  ; stosb также инкрементирует DI, так что DI теперь указывает на следующую
   ; точку в VGA-буфере
79  ; всегда переходим, так что это вечный цикл
80  jmp FillLoop

```

#### Алгоритм:

- Переключаемся в режим VGA 320\*200 256 цветов.  $320 * 200 = 64000$  (0xFA00). Каждый пиксель кодируется одним байтом, так что размер буфера 0xFA00 байт.

Он адресуется здесь при помощи пары регистров ES:DI.

ES должен быть здесь 0xA000, потому что это сегментный адрес видеобуфера, но запись числа 0xA000 в ES потребует по крайней мере 4 байта (PUSH 0A000h / POP ES). О 16-битной модели памяти в MS-DOS, читайте больше тут: ?? (стр. ??).

Учитывая, что BX здесь 0, и Program Segment Prefix находится по нулевому адресу, 2-байтная инструкция LES AX, [BX] запишет 0x20CD в AX и 0x9FFF в ES.

Так что программа начнет рисовать на 16 пикселей (или байт) перед видеобуфером.

Но это MS-DOS, здесь нет защиты памяти, так что запись происходит в самый конец обычной памяти, а там, как правило, ничего важного нет.

Вот почему вы видите красную полосу шириной 16 пикселей справа. Вся картинка сдвинута налево на 16 пикселей. Это цена экономии 2-х байт.

- Вечный цикл, обрабатывающий каждый пиксель. Наверное, самый общий метод обойти все точки на экране это два цикла: один для X-координаты, второй для Y-координаты.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

---

Но тогда вам придется перемножать координаты для поиска байта в видеобуфере VGA. Автор этого демо решил сделать наоборот: перебирать все байты в видеобуфере при помощи одного цикла вместо двух и затем получать координаты текущей точки при помощи деления.

В итоге координаты такие: X в пределах  $-256..63$  и Y в пределах  $-100..99$ . Вы можете увидеть на скриншоте что картинка как бы сдвинута в правую часть экрана. Это потому что самая большая черная дыра в форме сердца обычно появляется на координатах 0,0 и они здесь сдвинуты вправо.

Мог ли автор просто отнять 160 от X, чтобы получилось значение в пределах  $-160..159$ ? Да, но инструкция SUB DX, 160 занимает 4 байта, тогда как DEC DH — 2 байта (которая отнимает  $0x100$  (256) от DX). Так что картинка сдвинута ценой экономии еще 2-х байт.

- Проверить, является ли текущая точка внутри множества Мандельброта. Алгоритм такой же, как и описанный здесь.
- Цикл организуется инструкцией LOOP, которая использует регистр CX как счетчик. Автор мог бы установить число итераций на какое-то число, но не сделал этого: потому что 320 уже находится в CX (было установлено на строке 35), и это итак подходящее число как число максимальных итераций.

Мы здесь экономим немного места, не загружая другое значение в регистр CX.

- Здесь используется IMUL вместо MUL, потому что мы работаем со знаковыми значениями: помните, что координаты 0,0 должны быть где-то рядом с центром экрана.

Тоже самое и с SAR (арифметический сдвиг для знаковых значений): она используется вместо SHR.

- Еще одна идея — это упростить проверку пределов. Нам бы пришлось проверять пару координат, т.е. две переменных. Что делает автор это трижды проверяет на переполнение: две операции возведения в квадрат и одно прибавление. Действительно, мы ведь используем 16-битные регистры, содержащие знаковые значения в пределах  $-32768..32767$ , так что если любая из координат больше чем 32767 в процессе умножения, точка однозначно вышла за пределы, и мы переходим на метку MandelBreak.
- Здесь также имеется деление на 64 (при помощи инструкции SAR). 64 задает масштаб.

Попробуйте увеличить значение и вы получите более увеличенную картинку, или уменьшить для меньшей.

- Мы находимся на метке MandelBreak, есть только две возможности попасть сюда: цикл закончился с  $CX=0$  (точка внутри множества Мандельброта); или потому что произошло переполнение (CX все еще содержит какое-то значение). Записываем 8-битную часть CX (CL) в видеобуфер. Палитра по умолчанию грубая, тем не менее, 0 это черный: поэтому видим

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

черные дыры в местах где точки внутри множества Мандельброта.

Палитру можно инициализировать в начале программы, но не забывайте, это всего лишь программа на 64 байта!

- Программа работает в вечном цикле, потому что дополнительная проверка, где остановится, или пользовательский интерфейс, это дополнительные инструкции.

Еще оптимизационные трюки:

- 1-байтная CWD используется здесь для обнуления DX вместо двухбайтной XOR DX, DX или даже трехбайтной MOV DX, 0.
- 1-байтная XCHG AX, CX используется вместо двухбайтной MOV AX, CX. Текущее значение в AX все равно уже не нужно.
- DI (позиция в видеобufferе) не инициализирована, и будет 0xFFFFE в начале<sup>50</sup>. Это нормально, потому что программа работает бесконечно для всех DI в пределах 0..0xFFFF, и пользователь не может увидеть, что работала началась за экраном (последний пиксель видеобufferа 320\*200 имеет адрес 0xF9FF).

Так что некоторая часть работы на самом деле происходит за экраном. А иначе понадобятся дополнительные инструкции для установки DI в 0; добавить проверку на конец буфера.

## Моя «исправленная» версия

Листинг 8.26: Моя «исправленная» версия

```

1  org 100h
2  mov al,13h
3  int 10h
4
5  ; установить палитру
6  mov dx, 3c8h
7  mov al, 0
8  out dx, al
9  mov cx, 100h
10 inc dx
11 l00:
12 mov al, cl
13 shl ax, 2
14 out dx, al ; красный
15 out dx, al ; зеленый
16 out dx, al ; синий
17 loop l00
18
19 push 0a000h
20 pop es
21
```

<sup>50</sup>Больше о состояниях регистров на старте: <https://code.google.com/p/corkami/wiki/InitialValues#DOS>

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

```
22 xor di, di
23
24 FillLoop:
25 cwd
26 mov ax,di
27 mov cx,320
28 div cx
29 sub ax,100
30 sub dx,160
31
32 xor bx,bx
33 xor si,si
34
35 MandelLoop:
36 mov bp,si
37 imul si,bx
38 add si,si
39 imul bx,bx
40 jo MandelBreak
41 imul bp,bp
42 jo MandelBreak
43 add bx,bp
44 jo MandelBreak
45 sub bx,bp
46 sub bx,bp
47
48 sar bx,6
49 add bx,dx
50 sar si,6
51 add si,ax
52
53 loop MandelLoop
54
55 MandelBreak:
56 xchg ax,cx
57 stosb
58 cmp di, 0FA00h
59 jb FillLoop
60
61 ; дождаться нажатия любой клавиши
62 xor ax,ax
63 int 16h
64 ; установить текстовый видеорежим
65 mov ax, 3
66 int 10h
67 ; выход
68 int 20h
```

Автор сих строк попытался исправить все эти странности: теперь палитра плавная черно-белая, видеобуфер на правильном месте (строки 19..20), картинка рисуется в центре экрана (строка 30), программа в итоге заканчивается и ждет, пока пользователь нажмет какую-нибудь клавишу (строки 58..68).

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Но теперь она намного больше: 105 байт (или 54 инструкции)

<sup>51</sup>.

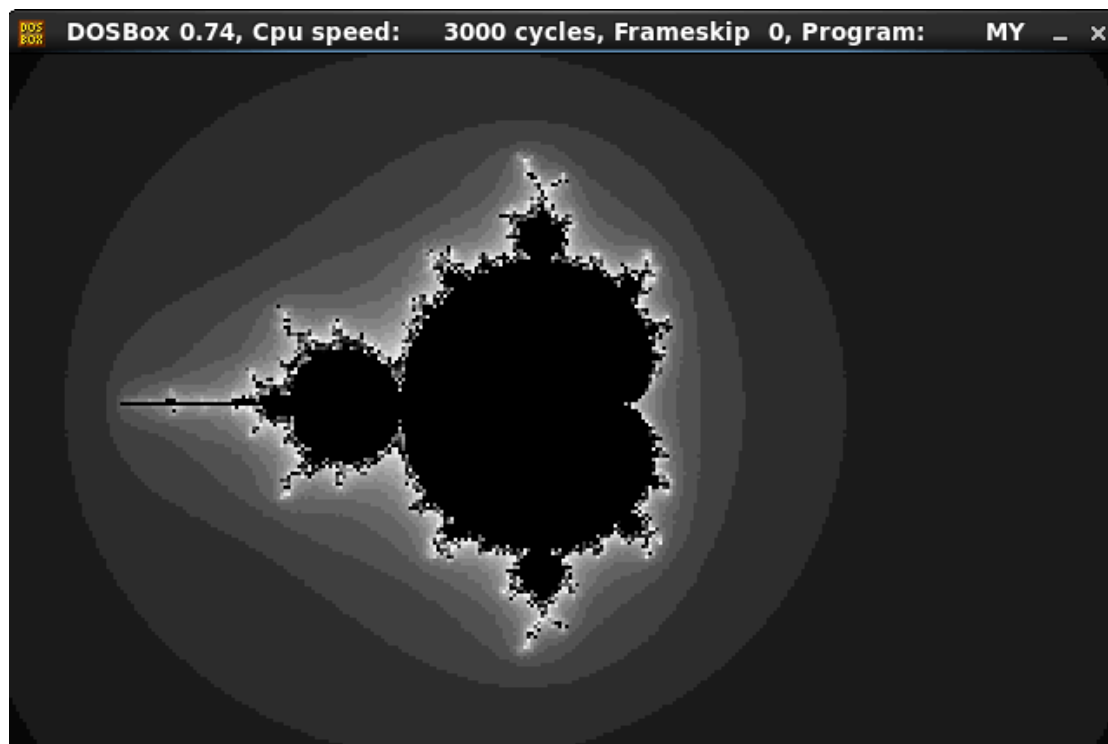


Рис. 8.18: Моя «исправленная» версия

Смотрите также: маленькая программа на Си печатающая множество Мандельброта в ASCII: [https://people.sc.fsu.edu/~jburkardt/c\\_src/mandelbrot\\_ascii/mandelbrot\\_ascii.html](https://people.sc.fsu.edu/~jburkardt/c_src/mandelbrot_ascii/mandelbrot_ascii.html)  
<https://miyuki.github.io/2017/10/04/gcc-archaeology-1.html>.

## 8.14. Как я переписывал 100 килобайт x86-кода на чистый Си

То была DLL-ка с секцией кода 100 килобайт, она брала на вход многоканальный сигнал и выдавала другой многоканальный сигнал. Там много всего было связано с обработкой сигналов. Внутри было очень много FPU-кода. Написано по-олдскульному, так, как писали в то время, когда передача параметров через аргументы ф-ций была дорогой, и потому использовалось много глобальных переменных и массивов, почти всё хранилось в них, а ф-ции, напротив,

<sup>51</sup> Можете поэкспериментировать и сами: скачайте DosBox и NASM и компилируйте так:  
`nasm file.asm -fbin -o file.com`

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

имели сравнительно мало аргументов, если вообще. Функции большие, их было около ста.

Тесты были, много.

Проблема была в том, что функции слишком большие и Hex-Rays неизменно выдавал немного неверный код. Нужно было очень внимательно всё чистить вручную. В процессе работы, я нашел в нем каких-то ошибок: ??.

Все 100 ф-ций декомпилировать сразу нельзя — где-то будут ошибки, тесты не пройдут, и где вы будете искать эти ошибки? Приходится переписывать по чуть-чуть.

В DLL-ке есть некая корневая ф-ция, скажем, `ProcessMain()`. Я переписываю её на Си при помощи Hex-Rays, она запускается из обычного .exe-процесса. Все ф-ции из DLL-ки, которые вызываются далее, у меня вызывались через указатели на ф-ции. DLL-ка загружена, и пока они все там.

**ASLR!** отключил, и DLL-ка каждый раз грузится по одному и тому же адресу, потому и адреса всех ф-ций одни и те же. Важно, что и адреса глобальных массивов тоже одни и те же

Затем переписываю ф-ции, вызывающиеся непосредственно из `ProcessMain()`, затем еще ниже, и т. д. Таким образом, ф-ции я постепенно перетаскивал из DLL в свою .exe. Каждый раз тестирую.

Много раз бывало и так — ф-ция слишком большая, например, несколько килобайт x86-кода, и после декомпиляции в Си, там что-то косячит, и неизвестно где. Из IDA я экспортировал её листинг в текст на ассемблере и компилировал при помощи обычного ассемблера (ML в MSVC). Она компилируется в .obj-файл и прикомпилируется к главной .exe, и пока всё ОК. Затем я делил эту ф-цию на более мелкие, здорово пригодился (когда бы еще?) опыт написания программ на чистом ассемблере в середине 90-х (руки до сих пор помнят). Если всё работает, более мелкие ф-ции постепенно переписывал на Си при помощи Hex-Rays, в то время как "головная" ф-ция более высокого уровня всё еще на ассемблере.

Интересно, что было много глобальных массивов, но границы между ними были сильно размыты. Но я вижу что есть какой-то большой кусок в секции .data, где лежит всё подряд. Дошел до стадии, когда на Си переписано уже всё, а все обращения к массивам происходят по адресам внутри секции .data в подгружаемой DLL-ке, впрочем, там почти не было констант. Затем, чтобы совсем отказаться от DLL-ки, я сделал большой глобальный "кусочек" уже у себя на Си, и вся работа с массивами шла через мой "кусочек", при том, что все массивы всё еще не были отделены друг от друга.

Вот реальный фрагмент оттуда, как было в начале. Значение — это адрес в .data-секции в DLL-ке:

```
int *a_val511=0x1002B588;
int *a_val483=0x1002B590;
int *a_val481=0x1002B5B8;
int *a_val515=0x1002B6E4;
...
```

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!



И все обращения происходят через указатели.

Потом я сделал "кусок":

```
char lump[0x1000000];

/* 0x1002B588 */int *a_val511=(int*)&lump[0x2B588];
/* 0x1002B590 */int *a_val483=(int*)&lump[0x2B590];
/* 0x1002B5B8 */int *a_val481=(int*)&lump[0x2B5B8];
/* 0x1002B6E4 */int *a_val515=(int*)&lump[0x2B6E4];
...
```

DLL-ку теперь можно было наконец-то отцепить и разбираться с границами массивов. Этот процесс я хотел немного автоматизировать и использовал для этого Pin. Я написал утилиту, которая показывала, по каким адресам в глобальном "куске" были обращения из каждого адреса. Точнее, в каких пределах? Так стало проще видеть границы массивов.

"На войне все средства хороши", так что я доходил и до того, что использовал Mathematica и Z3 для сокращения слишком длинных выражений (Hex-Rays не всё может оптимизировать):

[https://github.com/DennisYurichev/SAT\\_SMT\\_by\\_example/blob/master/proofs/simplify\\_EN.tex](https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/proofs/simplify_EN.tex).

Очень хорошим тестом было пересобрать всё под Linux при помощи GCC и заставить работать — как всегда, это было нелегко. Плюс, чтобы работало корректно и под x86 и под x64.

## 8.15. "Прикуп" в игре "Марьяж"

Знал бы прикуп — жил бы в Сочи.

Поговорка.

"Марьяж" — старая и довольно популярная версия игры в "Преферанс" под DOS.

Играют три игрока, каждому раздается по 10 карт, остальные 2 остаются в т.н. "прикупе". Начинаются торги, во время которых "прикуп" скрыт. Он открывается после того, как один из игроков сделает "заказ".

Знание карт в "прикупе" обычно имеет решающее преимущество.

Вот так в игре выглядит состояние "торгов", и "прикуп" посередине, скрытый:



Рис. 8.19: "Торги"

Попробуем "подсмотреть" карты в "прикупе" в этой игре.

Для начала — что мы знаем? Игра под DOS, датируется 1997-м годом. IDA показывает имена стандартных функций вроде `@GetImage$7Integert1t1t1m3Any` — это "манглинг" типичный для Borland Pascal, что позволяет сделать вывод, что сама игра написана на Паскале и скомпилирована Borland Pascal-ем.

Файлов около 10-и и некоторые имеют текстовую строку в заголовке "Marriage Image Library" — должно быть, это библиотеки спрайтов.

В IDA можно увидеть что используется функция `@PutImage$7Integert1m3Any4Word`, которая, собственно, рисует некий спрайт на экране. Она вызывается по крайней мере из 8-и мест. Чтобы узнать что происходит в каждом из этих 8-и мест, мы можем блокировать работу каждой функции и смотреть, что будет происходить. Например, первая ф-ция имеет адрес `seg002:062E`, и она заканчивается инструкцией `retf 0Eh` на `seg002:102A`. Это означает, что метод вызовов ф-ций в Borland Pascal под DOS схож с `stdcall` — вызываемая ф-ция должна сама возвращать стек в состояние до того как началась передача аргументов. В самом начале этой ф-ции вписываем инструкцию `"retf 0eh"`, либо 3 байта: `CA 0E 00`. Запускаем "Марьяж" и внешне вроде бы ничего не изменилось.

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

Переходим ко второй ф-ции, которая активно использует @PutImage\$q7Integert1m3Any4Word. Она находится по адресу seg008:0AB5 и заканчивается инструкцией retf 0Ah. Вписываем эту инструкцию в самом начале и запускаем:



Рис. 8.20: Карт нет

Карт не видно вообще. И видимо, эта функция их отображает, мы её заблокировали, и теперь карт не видно. Назовем эту ф-цию в IDA draw\_card(). Помимо @PutImage\$q7Integert1m3Any4Word, в этой ф-ции вызываются также ф-ции @SetColor\$q4Word, @SetFillStyle\$q4Wordt1, @Bar\$q7Integert1t1t1, @OutTextXY\$q7Integert16String.

Сама ф-ция draw\_cards() (её название мы дали ей сами только что) вызывается из 4-х мест. Попробуем точно также "блокировать" каждую ф-цию.

Когда я "блокирую" вторую, по адресу seg008:0DF3 и запускаю программу, вижу такое:

---

Если вы заметили опечатку, ошибку или имеете какие-то либо соображения, пожелания, пожалуйста, напишите мне: [мои адреса](#). Спасибо!

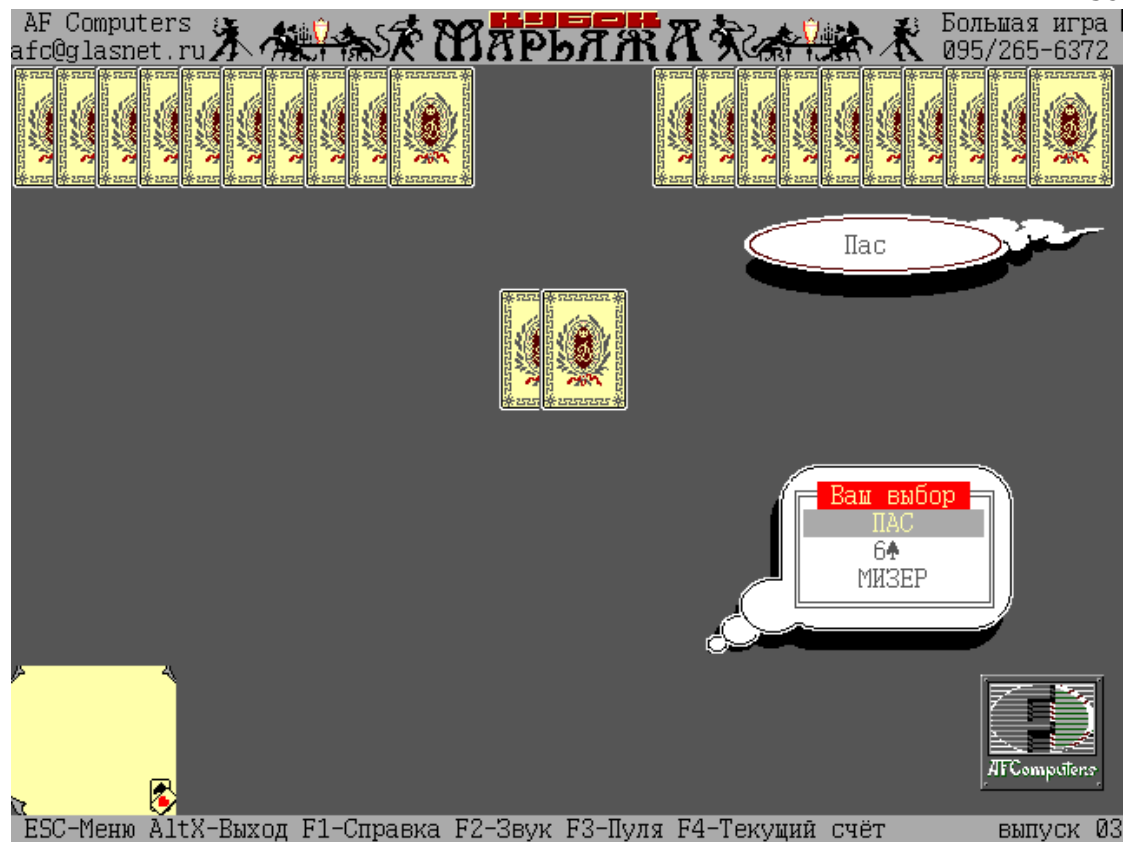


Рис. 8.21: Все карты кроме карт игрока

Видны все карты, кроме карт игрока. Видимо, эта функция рисует карты игрока.

Я переименовываю её в IDA в `draw_players_cards()`.

Четвертая ф-ция, вызывающая `draw_cards()`, находится по адресу `seg008:16B3`, и когда я её "блокирую", я вижу в игре такое:



**Ваш выбор**  
ПАС  
6♠  
МИЗЕР

Торговля  
0 0  
0

