Краткое введение в reverse engineering для начинающих

Денис Юричев <dennis@yurichev.com>

4 марта 2018 г.

Оглавление

Введение **iv**

2 Паттерны компиляторов **1**

2.1 Hello, world! . . . 2
2.2 Стек . . . 4
2.2.1 Сохранение адреса куда должно вернуться управление после вызова функции . . 4
2.2.2 Передача параметров для функции . . . 5
2.2.3 Хранение локальных переменных . . . 5
2.2.4 (Windows) SEH . . . 6
2.2.5 Защита от переполнений буфера . . . 7
2.3 printf() с несколькими аргументами . . . 8
2.4 scanf() . . . 10
2.4.1 Глобальные переменные . . . 12
2.4.2 Проверка результата scanf() . . . 13
2.5 Передача параметров через стек . . . 15
2.6 И еще немного о возвращаемых результатах . . . 17
2.7 Условные переходы . . . 18
2.8 switch()/case/default . . . 20
2.8.1 Если вариантов мало . . . 20
2.8.2 И если много . . . 21
2.9 Циклы . . . 25
2.10 strlen() . . . 28
2.11 Деление на 9 . . . 31
2.12 Работа с FPU . . . 33
2.12.1 Простой пример . . . 33
2.12.2 Передача чисел с плавающей запятой в аргументах . . . 35
2.12.3 Пример с сравнением . . . 36
2.13 Массивы . . . 41
2.13.1 Переполнение буфера . . . 42
2.13.2 Еще немного о массивах . . . 46
2.13.3 Многомерные массивы . . . 46
2.14 Битовые поля . . . 48
2.14.1 Проверка какого-либо бита . . . 48
2.14.2 Установка/сброс отдельного бита . . . 50
2.14.3 Сдвиги . . . 52
2.14.4 Пример вычисления CRC32 . . . 54
2.15 Структуры . . . 58
2.15.1 Пример SYSTEMTIME . . . 58
2.15.2 Выделяем место для структуры через malloc() . . . 59
2.15.3 Linux . . . 60
2.15.4 Упаковка полей в структуре . . . 61
2.15.5 Вложенные структуры . . . 63
2.15.6 Работа с битовыми полями в структуре . . . 64

2.16 Классы в Си++ . . . 70
2.16.1 GCC . . . 73
2.17 Указатели на функции . . . 75
2.17.1 GCC . . . 77
2.18 SIMD . . . 79
2.18.1 Векторизация . . . 79
2.18.2 Реализация `strlen()` () при помощи SIMD . . . 85
2.19 x86-64 . . . 89

3 Еще кое-что 96
3.1 Инструкция LEA . . . 97
3.2 Пролог и эпилог в функции . . . 98
3.3 npad . . . 99
3.4 Представление знака в числах . . . 101
3.4.1 Переполнение integer . . . 101
3.5 Способы передачи аргументов при вызове функций . . . 102
3.5.1 cdecl . . . 102
3.5.2 stdcall . . . 102
3.5.3 fastcall . . . 102
3.5.4 thiscall . . . 103
3.5.5 x86-64 . . . 103
3.5.6 Возвращение переменных типа *float*, *double* . . . 103

4 Поиск в коде того что нужно 104
4.1 Связь с внешним миром . . . 104
4.2 Строки . . . 105
4.3 Константы . . . 105
4.3.1 Magic numbers . . . 105
4.4 Поиск нужных инструкций . . . 106
4.5 Подозрительные паттерны кода . . . 108

5 Задачи 109
5.1 Легкий уровень . . . 109
5.1.1 Задача 1.1 . . . 109
5.1.2 Задача 1.2 . . . 110
5.1.3 Задача 1.3 . . . 112
5.1.4 Задача 1.4 . . . 113
5.1.5 Задача 1.5 . . . 115
5.1.6 Задача 1.6 . . . 116
5.1.7 Задача 1.7 . . . 117
5.1.8 Задача 1.8 . . . 118
5.1.9 Задача 1.9 . . . 118
5.1.10 Задача 1.10 . . . 119
5.2 Средний уровень . . . 119
5.2.1 Задача 2.1 . . . 119
5.3 crackme / keygenme . . . 126

6 Инструменты 127
6.0.1 Отладчик . . . 127

7 Что стоит почитать **128**
7.1 Книги . . . 128
7.1.1 Windows . . . 128
7.1.2 Си/Си++ . . . 128
7.1.3 x86 / x86-64 . . . 128
7.2 Блоги . . . 128
7.2.1 Windows . . . 128

8 Прочее **129**
8.1 Еще примеры . . . 129
8.2 Аномалии компиляторов . . . 129

9 Ответы на задачи **131**
9.1 Легкий уровень . . . 131
9.1.1 Задача 1.1 . . . 131
9.1.2 Задача 1.2 . . . 131
9.1.3 Задача 1.3 . . . 131
9.1.4 Задача 1.4 . . . 132
9.1.5 Задача 1.5 . . . 132
9.1.6 Задача 1.6 . . . 133
9.1.7 Задача 1.7 . . . 133
9.1.8 Задача 1.8 . . . 134
9.1.9 Задача 1.9 . . . 134
9.2 Средний уровень . . . 135
9.2.1 Задача 2.1 . . . 135

Введение

Здесь (будет) немного моих заметок о reverse engineering на русском языке для начинающих, для тех кто хочет научиться понимать создаваемый Си/Си++ компиляторами код, коего, практически, больше всего остального.

Наиболее используемых компилятора два: MSVC и GCC, на них и будем ставить эксперименты.

Имеется два основных синтаксиса X86 ассемблера: Intel (больше распространенный в DOS/Windows) и AT&T (распространен в *NIX) [1]. Здесь принят Intel-овский синтаксис. IDA 6 также выдает Intel-овский.

[1] http://en.wikipedia.org/wiki/X86_assembly_language#Syntax

Глава 2

Паттерны компиляторов

Когда я учил Си, а затем Си++, я просто писал небольшие куски кода, компилировал и смотрел что получилось на ассемблере, так мне понять было намного проще. Я делал это такое количество раз, что связь между кодом на Си/Си++ и тем что генерирует компилятор вбилась мне в подсознание достаточно глубоко, поэтому я могу глядя на код на асме сразу понимать, в общих чертах, что там было написано на Си. Возможно это поможет кому-то еще, попробую описать некоторые примеры.

2.1 Hello, world!

Начнем с знаменитого примера из книги "The C programming Language"[1]:

```
#include <stdio.h>

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

Компилируем в MSVC 2010: `cl 1.cpp /Fa1.asm`

(Ключ /Fa означает сгенерировать листинг на ассемблере)

```
CONST   SEGMENT
$SG3830 DB      'hello, world', 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
```

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*`, однако не имеет имени.

Но компилятору нужно как-то с ней работать, так что он дает ей внутреннее имя `$SG3830`.

Как видно, строка заканчивается нулевым байтом — это требования стандарта Си/Си++ насчет строк.

В сегменте кода `_TEXT` находится пока только одна функция — `_main`.

Функция `_main`, как и практически все функции, начинается с пролога и заканчивается эпилогом.

Об этом смотрите подробнее в разделе о прологе и эпилоге функции 3.2.

Далее следует вызов функции `printf()`: `CALL _printf`.

Перед этим вызовом, адрес строки (или указатель на нее) с нашим приветствием при помощи инструкции `PUSH` помещается в стек.

После того как функция `printf()` возвращает управление в функцию `main()`, адрес строки (или указатель на нее) все еще 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`.

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

Некоторые компиляторы, например Intel C++ Compiler, в этой же ситуации, могут вместо `ADD` сгенерировать `POP ECX` (это 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 против 3).

О стеке можно прочитать в соответствующем разделе 2.2.

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

[1] http://en.wikipedia.org/wiki/The_C_Programming_Language

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

`XOR`, на самом деле, как легко догадаться, "исключающее ИЛИ"[2], но компиляторы часто используют его вместо простого `MOV EAX, 0` — снова опкод немного короче (2 байта против 5).

Бывает так, что некоторые компиляторы генерируют `SUB EAX, EAX`, что значит, *отнять значение EAX от EAX*, в любом случае это даст 0 в результате.

Самая последняя инструкция `RET` возвращает управление в вызывающую функцию. Обычно, это код Си/Си++ CRT[3], который, в свою очередь, вернет управление операционной системе.

Теперь скомпилируем то же самое компилятором GCC 4.4.1 в Linux: `gcc 1.c -o 1`

Затем при помощи IDA 6. посмотрим как создалась функция `main()`.

С другой стороны, мы можем посмотреть результат работы GCC при помощи ключа `-S -masm=intel`)

```
main            proc near

var_10          = dword ptr -10h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     eax, offset aHelloWorld ; "hello, world"
                mov     [esp+10h+var_10], eax
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

Почти то же самое, за исключением того что в прологе функции мы видим `AND ESP, 0FFFFFFF0h` — эта инструкция выравнивает значение в `ESP` по 16-байтной границе, делая некоторые значения в стеке также выровненными по этой границе.

`SUB ESP, 10h` выделяет в стеке 16 байт, хотя, как будет видно далее, нам 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`.

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, ...`).

[2] http://en.wikipedia.org/wiki/Exclusive_or

[3] C Run-Time Code

2.2 Стек

Стек в компьютерных науках — это одна из наиболее фундаментальных вещей[4].

Технически, это просто кусок памяти процесса + регистр `ESP` который указывает где-то в пределах этого куска.

Часто используемые инструкции для работы со стеком это `PUSH` и `POP`. `PUSH` уменьшает `ESP` на 4, затем записывает по адресу на который указывает `ESP` содержимое своего единственного операнда.

`POP` это обратная операция — сначала достает из `ESP` значение и кладет его в операнд (который очень часто является регистром) и затем увеличивает `ESP` на 4. Конечно, это для 32-битной среды. В x64-среде это будет 8 а не 4.

В самом начале, `ESP` указывает на конец стека. `PUSH` уменьшает `ESP`, а `POP` — увеличивает. Конец стека находится в начале блока памяти. Это странно, но это так.

Для чего используется стек?

2.2.1 Сохранение адреса куда должно вернуться управление после вызова функции

При вызове другой функции через `CALL`, сначала в стек записывается адрес указывающий на место аккурат после инструкции `CALL`, затем делается безусловный переход (`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 80x86
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
; File c:\tmp6\ss.cpp
; Line 2
        push    ebp
        mov     ebp, esp
; Line 3
        call    ?f@@YAXXZ                               ; f
; Line 4
        pop     ebp
        ret     0
?f@@YAXXZ ENDP                                          ; f
```

... причем, если включить оптимизацию (`/Ox`), то будет даже интереснее, без переполнения стека, но работать будет *корректно*[5]:

```
?f@@YAXXZ PROC                                          ; f
; File c:\tmp6\ss.cpp
; Line 2
```

[4] http://en.wikipedia.org/wiki/Call_stack

[5] здесь ирония

```
$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.

2.2.2 Передача параметров для функции

```
push arg3
push arg2
push arg1
call f
add esp, 4*3
```

Вызываемая функция получает свои параметры также через указатель ESP.

См.также в соответствующем разделе о способах передачи аргументов через стек 3.5.

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.

К примеру, можно выделять в куче[6] место для аргументов, заполнять их и передавать в функцию указатель на это место через EAX. И это вполне будет работать.

Однако, так традиционно сложилось, что передача аргументов происходит именно через стек.

2.2.3 Хранение локальных переменных

Функция может выделить для себя некоторое место в стеке для локальных переменных просто отодвинув ESP глубже к концу стека.

Это снова не является необходимым требованием. Вы можете хранить локальные переменные где угодно. Но по традиции всё сложилось так.

Функция alloca()

Интересен случай с функцией alloca()[7].

Эта функция работает как malloc(), но выделяет память прямо в стеке.

Память освобождать через free() не нужно, так как эпилог функции 3.2 вернет ESP назад в изначальное состояние и выделенная память просто анулируется.

Интересна реализация функции alloca().

Эта функция, если упрощенно, просто сдвигает ESP вглубь стека на столько байт сколько вам нужно и возвращает ESP в качестве указателя на выделенный блок. Попробуем:

```
void f()
{
    char *buf=(char*)alloca (600);
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3);

    puts (buf);
};
```

Компилируем (MSVC 2010):

```
...

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

[6] heap в англоязычной литературе

[7] 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

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

    push    esi
    call    _puts
    add     esp, 28            ; 0000001cH

...
```

Единственный параметр в `alloca()` передается через `EAX`, а не как обычно через стек.

После вызова `alloca()`, `ESP` теперь указывает на блок в 600 байт который мы можем использовать под `buf`.

А GCC 4.4.1 обходится без вызова других функций:

```
                public f
f               proc near               ; CODE XREF: main+6

s               = dword ptr -10h
var_C           = dword ptr -0Ch

                push    ebp
                mov     ebp, esp
                sub     esp, 38h
                mov     eax, large gs:14h
                mov     [ebp+var_C], eax
                xor     eax, eax
                sub     esp, 624
                lea     eax, [esp+18h]
                add     eax, 0Fh
                shr     eax, 4          ; выровнять указатель
                shl     eax, 4          ; по 16-байтной границе
                mov     [ebp+s], eax
                mov     eax, offset format ; "hi! %d, %d, %d\n"
                mov     dword ptr [esp+14h], 3
                mov     dword ptr [esp+10h], 2
                mov     dword ptr [esp+0Ch], 1
                mov     [esp+8], eax    ; format
                mov     dword ptr [esp+4], 600 ; maxlen
                mov     eax, [ebp+s]
                mov     [esp], eax      ; s
                call    _snprintf
                mov     eax, [ebp+s]
                mov     [esp], eax      ; s
                call    _puts
                mov     eax, [ebp+var_C]
                xor     eax, large gs:14h
                jz      short locret_80484EB
                call    ___stack_chk_fail

locret_80484EB:                         ; CODE XREF: f+70
                leave
                retn
f               endp
```

2.2.4 (Windows) SEH

В стеке хранятся записи SEH (*Structured Exception Handling*) для функции (если имеются) [8].

[8] О SEH: классическая статья Мэтта Питрека: http://www.microsoft.com/msj/0197/Exception/Exception.aspx

2.2.5 Защита от переполнений буфера

More about it here 2.13.1.

2.3 printf() с несколькими агрументами

Попробуем теперь немного расширить пример *Hello, world!* 2.1, написав в теле функции `main()`:

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

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                                 ; 00000010H
```

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

Всего 4 аргумента. 4*4 = 16 — именно 16 байт занимают в стеке указатель на строку плюс еще 3 числа типа *int*.

Переменные типа *int* в 32-битной системе, как известно, имеет ширину 32 бита.

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.

См.также в соответствующем разделе о способах передачи аргументов через стек 3.5.

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
```

Скомпилируем то же самое в Linux при помощи GCC 4.4.1 и посмотрим в IDA 6 что вышло:

```
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
                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.

Кстати, эта разница неплохо иллюстрирует тот важный момент, что процессору, в общем, все равно как будут передаваться параметры функций. Можно создать гипотетический компилятор, который будет передавать их при помощи указателя на структуру с параметрами, не пользуясь стеком вообще.

2.4 scanf()

Теперь попробуем использовать scanf().

```
int main()
{
        int x;
        printf ("Enter X:\n");

        scanf ("%d", &x);

        printf ("You entered %d...\n", x);

        return 0;
};
```

Да, согласен, использовать `scanf()` в наши времена для того чтобы спросить у юзера что-то: не самая хорошая идея. Но я хотел проиллюстрировать передачу указателя на *int*.

Что получаем на ассемблере компилируя MSVC 2010:

```
CONST    SEGMENT
$SG3831    DB    'Enter X:', 0aH, 00H
    ORG $+2
$SG3832    DB    '%d', 00H
    ORG $+1
$SG3833    DB    'You entered %d...', 0aH, 00H
CONST    ENDS
PUBLIC    _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
_x$ = -4                        ; size = 4
_main    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    push   OFFSET $SG3831
    call   _printf
    add    esp, 4
    lea    eax, DWORD PTR _x$[ebp]
    push   eax
    push   OFFSET $SG3832
    call   _scanf
    add    esp, 8
    mov    ecx, DWORD PTR _x$[ebp]
    push   ecx
    push   OFFSET $SG3833
    call   _printf
    add    esp, 8
    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 ни откуда более. Так получилось, что локальные переменные располагаются в стеке. Может 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 сооветствующей 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.

Вообще, во все время исполнения функции, EBP указывает на текущий фрейм и через EBP+смещение можно иметь доступ как к локальным переменным функции, так и аргументам функции.

Можно было бы использовать ESP, но он во время исполнения функции постоянно меняется.

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$[ebp]. First of all, address of x is placed into EAX register by lea eax, DWORD PTR _x$[ebp] instruction

Инструкция LEA означает *load effective address*, но со временем она изменила свою функцию 3.1.

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 которого, строка: "You entered %d...\n".Второй аргумент: mov ecx, [ebp-4], эта инструкция кладет в ECX не адрес переменной x, а его значение, что там сейчас находится.

Далее значение ECX заталкивается в стек и вызывается последний printf().

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, 0FFFFFFF0h
                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
                leave
                retn
main            endp
```

GCC заменил первый вызов printf() на puts(). Действительно, printf() с одним агрументом 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.

Why GCC replaced printf() to puts? Because (puts()) work faster [9].

Видимо потому, что просто проталкивает символы в stdout не сравнивая каждый со знаком процента.

Почему scanf() переименовали в ___isoc99_scanf, я честно говоря, пока не знаю.

Далее все как и прежде — параметры заталкиваются через стек при помощи MOV.

[9] http://www.ciselant.de/projects/gcc_printf/gcc_printf.html

2.4.1 Глобальные переменные

А что если переменная `x` из предыдущего примера будет глобальной переменной а не локальной? Ну, тогда к ней смогут обращаться из любого другого места, а не только из тела функции. Это снова не очень хорошая практика программирования, но ради примера мы можем себе это позволить.

```
_DATA    SEGMENT
COMM    _x:DWORD
$SG2456    DB    'Enter X:', 0aH, 00H
    ORG $+2
$SG2457    DB    '%d', 00H
    ORG $+1
$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
    push   eax
    push   OFFSET $SG2458
    call   _printf
    add    esp, 8
    xor    eax, eax
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
```

Ничего особенного, в целом. Теперь `x` объявлена в сегменте `_DATA`. Память для нее в стеке более не выделяется. Все обращения к ней происходит не через стек, а уже напрямую. Её значение неопределено. Это означает, что память под нее будет выделена, но ни компилятор, ни ОС не будет заботиться о том, что там будет лежать на момент старта функции `_main`. В качестве домашнего задания, попробуйте объявить большой неопределенный массив и посмотреть что там будет лежать после загрузки.

Попробуем изменить объявление этой переменной:

```
int x=10; // default value
```

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 6, то увидите что x находится аккурат в начале сегмента `_DATA`, после этой переменной будут текстовые строки.

А вот если вы откроете в IDA 6, .exe скомплированный в прошлом примере, где значение x неопределено, то в IDA вы увидите:

```
.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
```

`_x` обозначен как `?`, наряду с другими переменными не требующими инициализции. Это означает, что при загрузке .exe в память, место под все это выделено будет. Но в самом .exe ничего этого нет. Неинициализированные переменные не занимают места в исполняемых файлах. Удобно для больших массивов, например.

В Linux все также почти. За исключением того что если значение `x` не определено, то эта переменная будет находится в сегменте `_bss`. В ELF[10] этот сегмент имеет такие аттрибуты:

```
; Segment type: Uninitialized
; Segment permissions: Read/Write
```

Ну а если сделать присвоение этой переменной значения 10, то она будет находится в сегменте `_data`, это сегмент с такими аттрибутами:

```
; Segment type: Pure data
; Segment permissions: Read/Write
```

2.4.2 Проверка результата scanf()

Как я уже говорил, использовать `scanf()` в наше время это слегка старомодно. Но если уж жизнь заставила этим заниматься, нужно хотя бы проверять, сработал ли `scanf()` правильно или пользователь ввел вместо числа что-то другое, что `scanf()` не смог трактовать как число.

```
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;
};
```

По стандарту, `scanf()`[11] возвращает количество успешно полученных значений.

В нашем случае, если все успешно и пользователь ввел таки некое число, `scanf()` вернет 1. А если нет, то 0 или EOF.

Мы добавили код, который проверяет результат `scanf()` и в случае ошибки, говорит пользователю что-то другое.

Вот, что выходит на ассемблере (MSVC 2010):

```
; Line 8
        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
```

[10] Формат исполняемых файлов, использующийся в Linux и некоторых других *NIX

[11] MSDN: scanf, wscanf

```
; Line 9
        mov     ecx, DWORD PTR _x$[ebp]
        push    ecx
        push    OFFSET $SG3834 ; 'You entered %d...', 0aH, 00H
        call    _printf
        add     esp, 8
; Line 10
        jmp     SHORT $LN1@main
$LN2@main:
; Line 11
        push    OFFSET $SG3836 ; 'What you entered? Huh?', 0aH, 00H
        call    _printf
        add     esp, 4
$LN1@main:
; Line 13
        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`.

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 перейти процессор по адресу указанном в операнде `JNE`, у нас это `$LN2@main`. Передав управление по этому адресу, процессор как раз начнет исполнять вызов `printf()` с аргументом `"What you entered? Huh?"`. Но если все нормально, перехода не случится, и исполнится другой `printf()` с двумя аргументами: `'You entered %d...'` и значением переменной `x`.

А для того чтобы после этого вызова не исполнился сразу второй вызов `printf()`, после него имеется инструкция `JMP`, безусловный переход, он отправит процессор на место аккурат после второго `printf()` и перед инструкцией `XOR EAX, EAX`, которая собственно `return 0`.

Итак, можно сказать, что в подавляющем случае 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[12]. `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 еденицы отнимется единица, получится ноль, и выставится флаг `ZF` (*zero flag*), означающий что последний полученный результат является нулем. Ни при каких других значениях `EAX`, флаг `ZF` выставлен не будет, 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 все будет работать также, но разница в том что `SUB` все-таки испортит значение в первом операнде. `CMP` это *SUB без сохранения результата*.

Код созданный при помощи GCC 4.4.1 в Linux практически такой же, если не считать мелких отличий, которые мы уже рассмотрели раннее.

[12] См.также о флагах x86-процессора: http://en.wikipedia.org/wiki/FLAGS_register_(computing).

2.5 Передача параметров через стек

Как мы уже успели заметить, вызывающая функция передает аргументы для вызываемой через стек. А как вызываемая функция имеет к ним доступ?

```
#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;
};
```

Имеем в итоге (MSVC 2010 Express):

```
_TEXT   SEGMENT
_a$ = 8                                                 ; size = 4
_b$ = 12                                                ; size = 4
_c$ = 16                                                ; size = 4
_f      PROC
; File c:\...\1.c
; Line 4
        push    ebp
        mov     ebp, esp
; Line 5
        mov     eax, DWORD PTR _a$[ebp]
        imul    eax, DWORD PTR _b$[ebp]
        add     eax, DWORD PTR _c$[ebp]
; Line 6
        pop     ebp
        ret     0
_f      ENDP

_main   PROC
; Line 9
        push    ebp
        mov     ebp, esp
; Line 10
        push    3
        push    2
        push    1
        call    _f
        add     esp, 12                                 ; 0000000cH
        push    eax
        push    OFFSET $SG2463 ; '%d', 0aH, 00H
        call    _printf
        add     esp, 8
; Line 11
        xor     eax, eax
; Line 12
        pop     ebp
        ret     0
_main   ENDP
```

Итак, здесь видно: в функции `_main` заталкиваются три числа в стек и вызывается функиця `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 стека относительно `EBP`.

Далее все более-менее просто: значение a кладется в `EAX`. Далее `EAX` умножается при помощи ин-

струкции IMUL на то что лежит в _b, так в EAX остается произведение[13] 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 вызываемой функции — она возьмет значение из EAX и отправит его в printf(). Скомпилируем то же в GCC 4.4.1 и посмотрим результат в IDA:

```
                public f
f               proc near               ; CODE XREF: main+20

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]
                imul    eax, [ebp+arg_4]
                add     eax, [ebp+arg_8]
                pop     ebp
                retn
f               endp

                public main
main            proc near               ; DATA XREF: _start+17

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        ; char *
                mov     [esp+10h+var_8], 3
                mov     [esp+10h+var_C], 2
                mov     [esp+10h+var_10], 1
                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 раннее.

[13]результат умножения

2.6 И еще немного о возвращаемых результатах

Резльутат выполнения функции возвращается[14] через регистр EAX, а если результат типа байт, то в самой младшей части EAX — AL. Если функция возвращает число с плавающей запятой, то регистр FPU ST(0) будет использован.

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. 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 /Ox):

```
$T3853 = 8                      ; size = 4
_a$ = 12                        ; size = 4
?get_some_values@@YA?AUs@@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]
    add    ecx, 3
    mov    DWORD PTR [eax+4], edx
    mov    DWORD PTR [eax+8], ecx
    ret    0
?get_some_values@@YA?AUs@@H@Z ENDP            ; get_some_values
```

Имя внутреннего макроса для передачи указателя на структуру здесь это $T3853.

[14] См.также: MSDN: Return Values (C++)

2.7 Условные переходы

Об условных переходах.

```
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);
    return 0;
};
```

Имеем в итоге функцию `f_signed()`:

```
_a$ = 8                            ; size = 4
_b$ = 12                        ; size = 4
_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', 0aH, 00H
    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', 0aH, 00H
    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', 0aH, 00H
    call   _printf
    add    esp, 4
$LN4@f_signed:
    pop    ebp
    ret    0
_f_signed ENDP
```

Первая инструкция JLE значит *Jump if Larger or Equal*. То есть, 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 второй операнд больше первого или равен ему. Кстати, если все три условных перехода сработают, ни один `printf()` не вызовется. Но, без внешнего вмешательства, это, пожалуй, невозможно.

GCC 4.4.1 производит почти такой же код, за исключением `puts()` 2.4 вместо `printf()`.

Далее функция `f_unsigned()` скомпилированная GCC:

```
.globl f_unsigned
    .type    f_unsigned, @function
f_unsigned:
    push    ebp
    mov     ebp, esp
    sub     esp, 24
    mov     eax, DWORD PTR [ebp+8]
    cmp     eax, DWORD PTR [ebp+12]
    jbe     .L7
    mov     DWORD PTR [esp], OFFSET FLAT:.LC0 ; "a>b"
    call    puts
.L7:
    mov     eax, DWORD PTR [ebp+8]
    cmp     eax, DWORD PTR [ebp+12]
    jne     .L8
    mov     DWORD PTR [esp], OFFSET FLAT:.LC1 ; "a==b"
    call    puts
.L8:
    mov     eax, DWORD PTR [ebp+8]
    cmp     eax, DWORD PTR [ebp+12]
    jae     .L10
    mov     DWORD PTR [esp], OFFSET FLAT:.LC2 ; "a<b"
    call    puts
.L10:
    leave
    ret
```

Здесь все то же самое, только инструкции условных переходов немного другие: `JBE` — *Jump if Below or Equal* и `JAE` — *Jump if Above or Equal*. Эти инструкции (`JA`/`JAE`/`JBE`/`JBE`) отличаются от `JG`/`JGE`/`JL`/`JLE` тем, что работают с беззнаковыми переменными.

Отступление: представление знака в числах 3.4. Таким образом, увидев где используется `JG`/`JL` вместо `JA`/`JBE` 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:

```
main:
    push    ebp
    mov     ebp, esp
    and     esp, -16
    sub     esp, 16
    mov     DWORD PTR [esp+4], 2
    mov     DWORD PTR [esp], 1
    call    f_signed
    mov     DWORD PTR [esp+4], 2
    mov     DWORD PTR [esp], 1
    call    f_unsigned
    mov     eax, 0
    leave
    ret
```

2.8 switch()/case/default

2.8.1 Если вариантов мало

```
void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    default: printf ("something unknown\n"); break;
    };
};
```

Это дает в итоге (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:
    mov    esp, ebp
    pop    ebp
    ret    0
_f    ENDP
```

В принципе, ничего особо нового для нас здесь, за исключением того, что компилятор зачем-то перекладывает входящую переменную `a` во временную в локальном стеке `v64`.

Если скомпилировать это при помощи GCC 4.4.1, то будет почти то же самое, даже с максимальной оптимизацией (ключ `-O3`).

Попробуем, включить оптимизацию кодегенератора MSVC (`/Ox`): `cl 1.c /Fa1.asm /Ox`

```
_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 $LN2@f
    mov     DWORD PTR _a$[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 все немного по-другому, причем не без грязных хаков.

Первое: `a` ложится 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 отнимания нуля от числа стал нулем) 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 образуется в результате 0, то сработает соответствующий переход.

И в конце концов, если ни один из условных переходов не сработал, управление передается `printf()` с агрументом `'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 затакливает в стек адрес возврата 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 адрес возврата
- 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 другой 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 адрес возврата и управление передается в ту функцию, которая вызывала `f()`, минуя при этом саму `f()`.

Все это возможно потому что `printf()` вызывается в `f()` в самом конце. Все это чем-то даже похоже на `longjmp()`[15]. И все это, разумеется, сделано для экономии времени исполнения.

2.8.2 И если много

А если ввтевлений слишком много, то конечно генерировать слишком длинный код с многочисленными `JE`/`JNE` уже не так удобно.

```
void f (int a)
{
    switch (a)
    {
```

[15] http://en.wikipedia.org/wiki/Setjmp.h

```
    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;
    };
};
```

Имеем в итоге (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
    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
    npad    2
$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 придумал им

компилятор. Все эти метки складываются во внутреннюю таблицу `$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. 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.

А там вызывается `printf()` с агрументом `'two'`. Дословно, инструкция `jmp DWORD PTR $LN11@f[ecx*4]` означает *перейти по DWORD, который лежит по адресу* `$LN11@f + ecx * 4`.

`npad` 3.3 это команда ассемблеру немного выровнять начало таблицы, дабы она располагалась по адресу кратному 4 (или 16). Это нужно для того чтобы процессор мог эффективнее загружать 32-битное значение из памяти, через шину с памятью, кеширование, итд.

Посмотрим что сгенерирует 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        ; char *
                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

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) 2.14.3. Затем адрес метки внутри функции берется из массива `off_804855C` и адресуется при помощи вычисленного индекса.

2.9 Циклы

Для организации циклов, в архитектуре x86 есть старая инструкция LOOP, она проверяет значение регистра ECX и если оно не ноль, делает декремент ECX и переход по метке указанной в операнде. Возможно, эта инструкция не слишком удобная, поэтому я не видел современных компиляторов, которые использовали бы её. Так что, если вы видите где-то LOOP, то это, с большой вероятностью, написанный руками код на ассемблере.

Кстати, в качестве домашнего задания, вы можете попытаться объяснить, чем именно эта инструкция неудобна.

Циклы на Си/Си++ описываются при помощи for(), while(), do/while().

Начнем с for().

Это выражение описывает инициализацию, условие, что делать после каждой итерации (инкремент/декремент) и тело цикла.

```
{
 for (инициализация; условие; после каждой итерации)
  тело_цикла;
}
```

Примерно также, генерируемый код и будет состоять из этих четырех частей.

Возьмем пример:

```
int main()
{
        int i;

        for (i=2; i<10; i++)
                f(i);

        return 0;
};
```

Имеем в итоге (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 f(i)
    push   ecx
    call   _f
    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:

```
main            proc near               ; DATA XREF: _start+17
```

```
var_20          = dword ptr -20h
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 20h
                mov     [esp+20h+var_4], 2  ; i initializing
                jmp     short loc_8048476

loc_8048465:
                mov     eax, [esp+20h+var_4]
                mov     [esp+20h+var_20], eax
                call    f
                add     [esp+20h+var_4], 1  ; i increment

loc_8048476:
                cmp     [esp+20h+var_4], 9
                jle     short loc_8048465   ; if i<=9, continue loop
                mov     eax, 0
                leave
                retn
main            endp
```

Интересно становится, если скомпилируем этот же код при помощи MSVC 2010 с включенной оптимизацией (`/Ox`):

```
_main    PROC
    push   esi
    mov    esi, 2
$LL3@main:
    push   esi
    call   _f
    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`.

В принципе, все 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 также 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`):

```
main            proc near               ; DATA XREF: _start+17

var_10          = dword ptr -10h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     [esp+10h+var_10], 2
                call    f
                mov     [esp+10h+var_10], 3
                call    f
                mov     [esp+10h+var_10], 4
                call    f
                mov     [esp+10h+var_10], 5
```

```
                call    f
                mov     [esp+10h+var_10], 6
                call    f
                mov     [esp+10h+var_10], 7
                call    f
                mov     [esp+10h+var_10], 8
                call    f
                mov     [esp+10h+var_10], 9
                call    f
                xor     eax, eax
                leave
                retn
main            endp
```

Однако, GCC просто *развернул* цикл[16].

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.

OK, увеличим максимальное значение i в цикле до 100 и попробуем снова. GCC выдаст подобное:

```
                public main
main            proc near

var_20          = dword ptr -20h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                push    ebx
                mov     ebx, 2              ; i=2
                sub     esp, 1Ch
                nop                         ; aligning label loc_80484D0 (loop body
                    begin) by 16-byte border

loc_80484D0:
                mov     [esp+20h+var_20], ebx ; pass i as first argument to f()
                add     ebx, 1              ; i++
                call    f
                cmp     ebx, 64h            ; i==100?
                jnz     short loc_80484D0   ; if not, continue
                add     esp, 1Ch
                xor     eax, eax            ; return 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 прийдется там сделать, то его значение будет сохранено в начале функции, прямо как в `main()` здесь.

[16] loop unwinding в англоязчной литературе

2.10 strlen()

Еще немного о циклах. Часто, функция `strlen()`[17] реализуется при помощи `while()`. Например, как это сделано в стандартных библиотеках MSVC:

```
int strlen (const char * str)
{
        const char *eos = str;

        while( *eos++ ) ;

        return( eos - str - 1 );
}
```

Итак, компилируем:

```
_eos$ = -4                          ; size = 4
_str$ = 8                           ; size = 4
_strlen PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _str$[ebp]    ; взять указатель на символ из str
    mov     DWORD PTR _eos$[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 с signed-расширением

    movsx   edx, BYTE PTR [ecx]
    mov     eax, DWORD PTR _eos$[ebp]    ; eax=eos
    add     eax, 1                       ; увеличить eax на еденицу
    mov     DWORD PTR _eos$[ebp], eax    ; положить eax назад в eos
    test    edx, edx                     ; edx==0?
    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 еденицу и возвращаем
       результат
    mov    esp, ebp
    pop    ebp
    ret    0
_strlen_ ENDP
```

Здесь две новых инструкции: `MOVSX` 2.10 и `TEST`.

О первой: `MOVSX` 2.10 предназначен 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. `MOVSX` 2.10 означает *MOV with Sign-Extent*. Оставшиеся биты с 8-го по 31-й `MOVSX` 2.10 сделает еденицей, если исходный байт в памяти имеет знак *минус*, или заполнит нулями, если знак *плюс*.

И вот зачем все это.

По стандарту Си/Си++, 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 0xFFFFFFFE. И 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 `MOVSX` 2.10.

См.также об этом раздел *Представление знака в числах* 3.4.

[17] подсчет длины строки в Си

Хотя, конкретно здесь, компилятору врядли 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 2.14. Но конкретно здесь, эта инструкция просто проверяет состояние регистра `EDX` на 0.

Попробуем GCC 4.4.1:

```
                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, вот только здесь используется `MOVZX` а не `MOVSX` 2.10. `MOVZX` означает *MOV with Zero-Extent*. Эта инструкция перекладывает какое-либо значение в регистр и остальное добивает нулями. 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 *not zero*) выставит еденицу в `AL`. Смысл этой процедуры в том, что, если говорить человеческим языком, *если AL не ноль, то выполнить переход на* `loc_80483F0`. Компилятор выдал немного избыточный код, но не будем забывать что оптимизация выключена.

Теперь скомпилируем все то же самое в MSVC 2010, но с включенной оптимизацией (`/Ox`):

```
п»ї_str$ = 8                        ; size = 4
_strlen PROC
    mov     ecx, DWORD PTR _str$[esp-4]   ; ECX -> pointer to the string
    mov     eax, ecx                      ; move to EAX
$LL2@strlen_:
    mov     dl, BYTE PTR [eax]            ; DL = *EAX
    inc     eax                           ; EAX++
    test     dl, dl                       ; DL==0?
    jne     SHORT $LL2@strlen_            ; no, continue loop
    sub     eax, ecx                      ; calculate pointers difference
```

```
    dec     eax                           ; decrement EAX
    ret     0
_strlen_ ENDP
```

Здесь все попроще стало. Но следует отметить, что компилятор обычно может так хорошо использовать регистры только на не очень больших функций с не очень большим количеством локальных переменных.

INC/DEC — это инструкции инкремента-декремента, попросту говоря: увеличить на еденицу или уменьшить.

Попробуем GCC 4.4.1 с влюченной 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
                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, только кроме того что почему-то используется MOVZX, который 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 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, 0ffffffffh`. 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.

См. также раздел: Представление знака в числах 3.4.

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? Снова не берусь сказать. Но я не сомневаюсь, что эти оба варианта работают примерно равноценно в плане эффективности и скорости.

2.11 Деление на 9

Простая функция:

```
int f(int a)
{
        return a/9;
};
```

Компилируется вполне предсказуемо:

```
_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
    ret    0
_f   ENDP
```

IDIV делит 64-битное число хранящееся в паре регистров EDX:EAX на значение в ECX. В результате, EAX будет содержать результат деления, 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, также как это делает MOVSX 2.10. Со включенной оптимизацией (/Ox) получается:

```
_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 [18] создать код эквивалентный тому что мы хотим и работающий быстрее. GCC 4.4.1 даже без включенной оптимизации генерит примерно такой же код как и MSVC с оптимизацией:

```
       public f
f      proc near

arg_0  = dword ptr  8

       push    ebp
       mov     ebp, esp
       mov     ecx, [ebp+arg_0]
       mov     edx, 954437177
       mov     eax, ecx
       imul    edx
       sar     edx, 1
       mov     eax, ecx
       sar     eax, 1Fh
       mov     ecx, edx
       sub     ecx, eax
       mov     eax, ecx
```

[18] Подробнее о делении через умножение: MSDN: Integer division by constants, http://www.nynaeve.net/?p=115

```
        pop     ebp
        retn
f       endp
```

2.12 Работа с FPU

FPU (*Floating-point unit*) — девайс в процессоре работающий с числами с плавающей запятой.

Раньше он назывался сопроцессором. Он немного похож на программируемый калькулятор и стоит немного в стороне от основного процессора.

Перед изучением FPU полезно ознакомиться с тем как работают стековые машины[19], или ознакомиться с основами языка Forth[20].

Интересен факт, что в свое время (до 80486) сопроцессор был отдельным чипом на материнской плате, и вследствии его высокой цены, он стоял не всегда. Его можно было докупить отдельно и поставить.

Начиная с процессора 80486, FPU уже всегда входит в его состав.

FPU имеет стек из восьми 80-битных регистров, каждый может содержать число в формате IEEE 754[21].

В Си/Си++ типы имеются два типа для работы с числами с плавающей запятой, это *float* (*число одинарной точности*[22], 32 бита) [23] и *double* (*число двойной точности*[24], 64 бита).

GCC поддерживает тип *long double* (*extended precision*[25], 80 бит), но MSVC — нет.

Не смотря на то что *float* занимает столько же места сколько *int* на 32-битной архитектуре, представление чисел, разумеется, совершенно другое.

Число с плавающей точкой состоит из знака, мантиссы[26] и экспоненты.

Функция, имеющая *float* или *double* среди аргументов, получает эти значения через стек. Если функция возвращает *float* или *double*, она оставляет значение в регистре `ST(0)` — то есть, на вершине FPU-стека.

2.12.1 Простой пример

Рассмотрим простой пример:

```
double f (double a, double b)
{
        return a/3.14 + b*4.1;
};
```

Компилим в MSVC 2010:

```
CONST    SEGMENT
__real@4010666666666666 DQ 04010666666666666r    ; 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]

; состояние стека сейчас: ST(0) = _a

    fdiv   QWORD PTR __real@40091eb851eb851f

; состояние стека сейчас: ST(0) = результат деления _a на 3.13
```

[19] http://en.wikipedia.org/wiki/Stack_machine
[20] http://en.wikipedia.org/wiki/Forth_(programming_language)
[21] http://en.wikipedia.org/wiki/IEEE_754-2008
[22] http://en.wikipedia.org/wiki/Single-precision_floating-point_format
[23] Формат представления float-чисел затрагивается в разделе *Работа с типом float как со структурой* 2.15.6.
[24] http://en.wikipedia.org/wiki/Double-precision_floating-point_format
[25] http://en.wikipedia.org/wiki/Extended_precision
[26] *significand* или *fraction* в анлоязчной литературе

```
    fld     QWORD PTR _b$[ebp]

; состояние стека сейчас: ST(0) = _b; ST(1) = результат деления _a на 3.13

    fmul    QWORD PTR __real@4010666666666666

; состояние стека сейчас: ST(0) = результат _b * 4.1; ST(1) = результат деления _a на 3.13

    faddp   ST(1), ST(0)

; состояние стека сейчас: ST(0) = результат сложения

    pop     ebp
    ret     0
_f   ENDP
```

FLD берет 8 байт из стека и загружает из 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, так что то что мы там видим, это шестандцатиричное представление числа *3.14* в формате IEEE 754.

После выполнения FDIV, в ST(0) остается результат деления.

Кстати, есть еще инструкция FDIVP, которая делит ST(1) на ST(0), выталкивает эти числа из стека и заталкивает результат. Если вы знаете язык Forth[27], то это как раз оно и есть — стековая машина[28].

Следующая FLD заталкивает в стек значение b.

После этого, в ST(1) перемещается результат деления, а в ST(0) теперь будет 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.

GCC 4.4.1 (с опцией -O3) генерирует похожий код, хотя и с некоторой разницей:

```
                public f
f               proc near

arg_0           = qword ptr  8
arg_8           = qword ptr  10h

                push    ebp
                fld     ds:dbl_8048608 ; 3.14

; состояние стека сейчас: ST(0) = 3.13

                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
```

[27] http://en.wikipedia.org/wiki/Forth_(programming_language)

[28] http://en.wikipedia.org/wiki/Stack_machine

```
                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).

FDIVR означает *Reverse Divide* — делить поменяв делитель и делимое местами. Точно такой же инструкции для умножения нет, потому она была бы бессмыслена (ведь умножение — операция коммутативная), так что там остается только FMUL без соответсвующей 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.

Этот кусок кода получен при помощи IDA 6, которая регистр ST(0) называет для краткости просто ST.

2.12.2 Передача чисел с плавающей запятой в аргументах

```
int main ()
{
        printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

        return 0;
}
```

Посмотрим что у нас вышло (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)

    fstp   QWORD PTR [esp] ; перегрузить результат из ST(0) в локальный стек для printf()
    push   OFFSET $SG2651
    call   _printf
    add    esp, 12
    xor    eax, eax
    pop    ebp
    ret    0
_main    ENDP
```

FLD и FSTP перемешают переменные из/в сегмента данных в FPU-стек. pow()[29] достает оба значения из FPU-стека и возвращает результат в ST(0). printf() берет 8 байт из стека и трактует их как переменную типа *double*.

[29]стандартная функция Си, возводящая число в степень

2.12.3 Пример с сравнением

Попробуем теперь вот это:

```
double d_max (double a, double b)
{
        if (a>b)
                return a;

        return b;
};
```

Несмотря на кажущуюся простоту этой функции, понять как она работает будет чуть сложнее.

Вот что выдал 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

; we are here if a>b

    fld    QWORD PTR _a$[ebp]
    jmp    SHORT $LN2@d_max
$LN1@d_max:
    fld    QWORD PTR _b$[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.

Итак, биты `C3`/`C2`/`C0` выставлены, но, к сожалению, у процессоров до Intel P6 [30] нет инструкций условного перехода, проверяющих эти биты. Возможно это исторически так сложилось (вспомните о том что FPU когда-то был вообще отдельным чипом). А у Intel P6 появились инструкции `FCOMI`/`FCOMIP`/`FUCOMI`/`FUCOMIP` — делающие тоже самое, только напрямую модифицирующие флаги `ZF`/`PF`/`CF`.

После этого, инструкция `FCOMP` выдергивает одно значение из стека. Это отличает её от `FCOM`, которая просто сравнивает значения, оставляя стек в таком же состоянии.

`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.

[30] Intel P6 это Pentium Pro, Pentium II, и далее

- Если a>b, то биты будут выставлены: 0, 0, 1.
- Если a=b, то биты будут выставлены так: 1, 0, 0.

После исполнения `test ah, 5`, бит C3 и C1 сбросится в ноль, на позициях 0 и 2 (внутри регистра AH) останутся соответственно C0 и C2.

Теперь немного о *parity flag*[31]. Еще один замечательный рудимент:

> 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 can not 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.[32]

Этот флаг выставляется в 1 если количество едениц в последнем результате — четно. И в ноль если — нечетно.

Таким образом, что мы имеем, флаг PF будет выставлен в еденицу, если C0 и C2 оба нули или оба единицы. 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 уже *вылетел* после исполнения `test ah, 5`).

Дальше все 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.

Но это еще не все!

А теперь скомпилим все это в MSVC 2010 с опцией /Ox

```
_a$ = 8             ; size = 8
_b$ = 16            ; size = 8
_d_max      PROC
    fld     QWORD PTR _b$[esp-4]
    fld     QWORD PTR _a$[esp-4]

; состояне 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

; 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
```

[31] флаг четности

FCOM отличается от FCOMP тем что просто сравнивает значения и оставляет стек в том же состоянии. В отличие от предыдущего примера, операнды здесь в другом порядке. Поэтому и результат сравнения в C3/C2/C0 будет другим чем раньше:

- Если a>b в нашем случае, то биты C3/C2/C0 должны быть выставлены так: 0, 0, 0.
- Если b>a, то биты будут выставлены: 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 лежат _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), что 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 инструкции которая просто выдергивает значение из стека и больше ничего.

Но и это еще не все.

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
                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

                fucompp            ; 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 еденицу 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[33]. 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 "сигнализирующее нечисло".

Далее мы видим SAHF — это довольно редкая инструкция в коде не использущим FPU. 8 бит из AH перекладываются в младшие 8 бит регистра статуса процессора в таком порядке: `SF:ZF:-:AF:-:PF:-:CF <- AH`.

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` перекладывает 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: 0, 0, 1.
- Если a=b, то биты будут выставлены так: 1, 0, 0.

Иными словами, после инструкций FUCOMPP/FNSTSW/SAHF, мы получим такое состояние 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: `ZF=0, PF=0, CF=1`.
- Если a=b, то флаги будут выставлены 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[34] выставляет 1 или 0 в AL, а Jcc делает переход или нет. SETNBE запишет 1 если только 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 единица, последующий условный переход JZ взят не будет, и функция вернет _a. В остальных случаях, функция вернет _b.

Но и это еще не конец.

[33] http://ru.wikipedia.org/wiki/NaN
[34] *cc* это *condition code*

GCC 4.4.1 с оптимизацией -O3

```
                public d_max
d_max           proc near

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
                fstp    st ; записать ST(0) в ST(0) (холостая операция), выкинуть значение
                    лежащее на вершине стека, оставить _b
                jmp     short loc_804844A

loc_8048448:
                fstp    st(1) ; записать _a в ST(0), выкинуть значение лежащее на вершине
                   стека, оставить _a на вершине стека

loc_804844A:
                pop     ebp
                retn
d_max           endp
```

Почти все что здесь есть уже описано мною, кроме одного: использование JA после SAHF. Действительно, инструкции условных переходов "больше "меньше "равно"для сравнения беззнаковых чисел (JA, JAE, JBE, JBE, JE/JZ, JNA, JNAE, JNB, JNBE, JNE/JNZ) проверяют только флаги CF и ZF. И биты C3/C2/C0 после сравнения перекладываются в эти флаги аккурат так, чтобы перечисленные инструкции переходов могли работать. JA сработает если CF и ZF обнулены.

Таким образом, перечисленные инструкции условного перехода можно использовать после инструкций FNSTSW/SAHF.

Вполне возможно что биты статуса FPU C3/C2/C0 преднамерено были размещены таким образом, чтобы переноситься на базовые флаги процессора без перестановок.

2.13 Массивы

Массив это просто набор переменных в памяти, обязательно лежащих рядом, и обязательно одного типа.

```
#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;
};
```

Компилируем:

```
_TEXT    SEGMENT
_i$ = -84                        ; size = 4
_a$ = -80                        ; size = 80
_main      PROC
    push   ebp
    mov    ebp, esp
    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
```

Однако, ничего особенного, просто два цикла, один заполняет цикл, второй печатает его содержимое. Команда `shl ecx, 1` используется для умножения `ECX` на 2, об этом ниже 2.14.3.

Под массив выделено в стеке 80 байт, это 20 элементов по 4 байта.

То что делает GCC 4.4.1:

```
                public main
main            proc near               ; DATA XREF: _start+17

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, 0FFFFFFF0h
                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
```

2.13.1 Переполнение буфера

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 больше двадцати? Эта одна из особенностей Си/Си++, за которую их, собственно, и ругают.

Вот код который и компилится и работает:

```
#include <stdio.h>

int main()
{
        int a[20];
        int i;

        for (i=0; i<20; i++)
                a[i]=i*2;

        printf ("a[100]=%d\n", a[100]);

        return 0;
};
```

Вот в это (MSVC 2010):

```
_TEXT    SEGMENT
_i$ = -84                        ; size = 4
_a$ = -80                        ; size = 80
_main      PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 84           ; 00000054H
    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]
    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+400]
    push   eax
    push   OFFSET $SG2460
    call   _printf
    add    esp, 8
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main      ENDP
```

У меня оно при запуске выдало вот это:

```
a[100]=760826203
```

Это просто *что-то*, что волею случая лежало в стеке рядом с массивом, через 400 байт от его первого элемента.

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[35], что делало бы запускаемый код медленнее.

Итак, мы прочитали какое-то число из стека явно *нелегально*, а что если мы запишем?

Вот что мы пишем:

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

[35] Java, Python, итд

```
int main()
{
        int a[20];
        int i;

        for (i=0; i<30; i++)
                a[i]=i;

        return 0;
};
```

И вот что имеем на ассемблере:

```
_TEXT    SEGMENT
_i$ = -84                    ; size = 4
_a$ = -80                    ; size = 80
_main      PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 84                         ; 00000054H
    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.

Отладчик я уже давно не использую, так как надоело для всяких мелких задач вроде подсмотреть состояние регистров, запускать что-то, двигать мышью, итд. Поэтому я написал очень минималистическую утилиту для себя, *tracer* 6.0.1, коей обхожусь.

Помимо всего прочего, я могу использовать мою утилиту просто чтобы посмотреть где и какое исключение произошло. Итак, пробую:

```
generic tracer 0.4 (WIN32), http://conus.info/gt

New process: C:\PRJ\...\1.exe, PID=7988
EXCEPTION_ACCESS_VIOLATION: 0x15 (<symbol (0x15) is in unknown module>),
    ExceptionInformation[0]=8
EAX=0x00000000 EBX=0x7EFDE000 ECX=0x0000001D EDX=0x0000001D
ESI=0x00000000 EDI=0x00000000 EBP=0x00000014 ESP=0x0018FF48
EIP=0x00000015
FLAGS=PF ZF IF RF
PID=7988|Process exit, return code -1073740791
```

Итак, следите внимательно за регистрами.

Исключение произошло по адресу 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 — 0x1D.

И еще немного изучим разметку стека.

После того как управление передалось в _main, в стек было сохранено значение EBP. Затем, для массива + переменной i было выделено 84 байта. Это (20+1)*sizeof(int). ESP сейчас указывает на переменную _i в локальном стеке и при исполнении следующего PUSH что-либо, *что-либо* появится рядом с _i.

Вот так выглядит разметка стека пока управление находится внутри _main:

```
ESP:      4 байта для i
ESP+4:    80 байт для массива a[20]
ESP+84:   сохраненное EBP
ESP+88:   адрес возврата
```

Команда a[19]=чего_нибудь записывает последний *int* в пределах массива (пока что!)

Команда a[20]=чего_нибудь записывает *чего_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 все дело в том, что заканчиваясь, эпилог функции восстанавливал значение EBP. (20 в десятичной системе это как раз 0x14 в шестнадцетиричной). Далее выполнилась инструкция RET, которая на самом деле эквивалентна POP EIP.

Инструкция RET вытащила из стека адрес возврата (это адрес в какой-то CRT[36]-функции, которая вызвала _main), а там было записано 21 в десятичной системе, то есть 0x15 в шестнадцетиричной. И вот процессор оказался по адресу 0x15, но исполняемого кода там нет, так что случилось исключение.

Добро пожаловать! Это называется *buffer overflow*[37].

Замените массив *int* на строку (массив *char*), нарочно создайте слишком длинную строку, просуньте её в ту программу, в ту функцию, которая не проверяя длину строки скопирует её в слишком короткий буфер, и вы сможете указать программе, по какому именно адресу перейти. Не все так просто в реальности, конечно, но началось все с этого[38].

В наше время пытаются бороться с этой напастью, не взирая на халатность программистов на Си/Си++. В MSVC есть опции вроде[39]:

```
/RTCs Stack Frame runtime checking
/GZ Enable stack checks (/RTCs)
```

Один из методов, это в прологе функции вставлять в область локальных переменных некоторое случайное значение и в эпилоге функции, перед выходом, это число проверять. И если проверка не прошла, то не выполнять инструкцию RET а остановиться (или зависнуть). Процесс зависнет, но это лучше чем удаленная атака на ваш хост.

Попробуем то же самое в GCC 4.4.1. У нас выходит такое:

```
                public main
main            proc near

a               = dword ptr -54h
i               = dword ptr -4

                push    ebp
                mov     ebp, esp
                sub     esp, 60h
                mov     [ebp+i], 0
                jmp     short loc_80483D1
loc_80483C3:
                mov     eax, [ebp+i]
                mov     edx, [ebp+i]
                mov     [ebp+eax*4+a], edx
                add     [ebp+i], 1
```

[36]C Run-Time

[37]http://en.wikipedia.org/wiki/Stack_buffer_overflow

[38]Классическая статья об этом: Smashing The Stack For Fun And Profit

[39]Wikipedia: описания защит, которые компилятор может вставлять в код

```
loc_80483D1:
                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, это потому что разметка стека чуть другая.

2.13.2 Еще немного о массивах

Теперь понятно, почему нельзя написать в исходном коде на Си/Си++ что-то вроде: [40]

```
void f(int size)
{
    int a[size];
...
};
```

Все просто потому, чтобы выделять место под массив в локальном стеке или же сегменте данных (если массив глобальный), компилятору нужно знать его размер, чего он, на стадии компиляции, разумеется знать не может.

Если вам нужен массив произвольной длины, то выделите столько, сколько нужно, через `malloc()`, затем обращайтесь к выделенному блоку байт как к массиву того типа, который вам нужен.

2.13.3 Многомерные массивы

Многомерный массив выглядит внутри так же как и линейный.

Попробуем:

```
#include <stdio.h>

int a[10][20][30];
```

[40] GCC способен это сделать выделяя место под массив динамически в стеке (как alloca()), но это расширение не является частью стандарта

```
void insert(int x, int y, int z, int value)
{
        a[x][y][z]=value;
};
```

В итоге (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$[ebp]
    imul   eax, 2400                  ; 00000960H
    mov    ecx, DWORD PTR _y$[ebp]
    imul   ecx, 120                  ; 00000078H
    lea    edx, DWORD PTR _a[eax+ecx]
    mov    eax, DWORD PTR _z$[ebp]
    mov    ecx, DWORD PTR _value$[ebp]
    mov    DWORD PTR [edx+eax*4], ecx
    pop    ebp
    ret    0
_insert    ENDP
_TEXT      ENDS
```

В принципе, ничего удивительного. В `insert()` для индексирования нужного элемента массива, три входных аргумента перемножаются так, чтобы представить массив трехмерным.

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]
                mov     eax, edx
                shl     eax, 4
                sub     eax, edx
                imul    edx, ebx, 600
                add     eax, edx
                lea     edx, [eax+ecx]
                mov     eax, [ebp+value]
                mov     dword ptr ds:a[edx*4], eax
                pop     ebx
                pop     ebp
                retn
insert          endp
```

2.14 Битовые поля

Немало функций задают различные флаги в аргументах при помощи битовых полей[41]. Наверное, вместо этого, можно было бы использовать набор переменных типа *bool*, но это было бы не очень экономно.

2.14.1 Проверка какого-либо бита

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):

```
        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:

```
#define GENERIC_READ                     (0x80000000L)
#define GENERIC_WRITE                    (0x40000000L)
#define GENERIC_EXECUTE                  (0x20000000L)
#define GENERIC_ALL                      (0x10000000L)
```

Все ясно, `GENERIC_READ | GENERIC_WRITE = 0x80000000 | 0x40000000 = 0xC0000000`, и это значение используется как второй аргумент для `CreateFile()`[42] function.

Как `CreateFile()` будет проверять флаги?

Заглянем в KERNEL32.DLL от Windows XP SP3 x86 и найдем в функции `CreateFileW()` в том числе и такой кусок кода:

```
.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
```

Здесь мы видим инструкцию `TEST`, впрочем, она берет не весь второй аргумент функции, но 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 2.4.2).

Логика данного куска 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>
```

[41] bit fields в анлоязычной литературе
[42] MSDN: CreateFile function

```
void main()
{
        int handle;

        handle=open ("file", O_RDWR | O_CREAT);
};
```

Получим:

```
                public main
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 вызов сисколла:

```
.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()` вероятно 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.

Итак, при вызове сисколла `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`).

Важное отступление. Помимо передачи параметров функции через стек, существует также возможность передавать некоторые из них через регистры. Это называется в том числе fastcall 3.5.3. Это работает немного быстрее, так как процессору не нужно обращаться к стеку лежащему в памяти для чтения аргументов. В GCC есть опция *regparm*[43], и с её помощью можно задать, сколько аргументов можно передать через регистры.

Ядро Linux 2.6 собирается с опцией `-mregparm=3` [44] [45].

И 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 регистров.

Итак, качаем ядро 2.6.31, собираем его в Ubuntu: `make vmlinux`, открываем в IDA 6, находим функцию `do_filp_open()`. В начале мы увидим подобное (комментарии мои):

```
do_filp_open    proc near
...
                push    ebp
                mov     ebp, esp
                push    edi
                push    esi
```

[43] http://ohse.de/uwe/articles/gcc-attributes.html#func-regparm
[44] http://kernelnewbies.org/Linux_2_6_20#head-042c62f290834eb1fe0a1942bbf5bb9a4accbc8f
[45] См. также файл `arch\x86\include\asm\calling.h` в исходниках ядра

```
                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
                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 кусок:

```
loc_C01EF6B4:                           ; CODE XREF: do_filp_open+4F
                test    bl, 40h         ; O_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 то чему равен макрос `O_CREAT`. `open_flag` проверяется на наличие бита 0x40 и если бит равен 1, то выполняется следующие за `JNZ` инструкции.

2.14.2 Установка/сброс отдельного бита

Например:

```
#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;
};
```

Имеем в итоге (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$[ebp]
    and    edx, -513                   ; fffffdffH
    mov    DWORD PTR _rt$[ebp], edx
    mov    eax, DWORD PTR _rt$[ebp]
```

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

Инструкция `OR` здесь добавляет в переменную еще один бит, игнорируя остальные.

А `AND` сбрасывает некий бит. Можно также сказать, что `AND` здесь копирует все биты, кроме одного. Действительно, во втором операнде `AND` выставлены в еденицу те биты, которые нужно сохранить, кроме одного, копировать который мы не хотим (и который 0 в битовой маске). Так проще понять и запомнить.

Если скомпилировать в MSVC с оптимизацией (`/Ox`), то код будет еще короче:

```
_a$ = 8                      ; size = 4
_f    PROC
    mov     eax, DWORD PTR _a$[esp-4]
    and     eax, -513          ; fffffdffH
    or      eax, 16384         ; 00004000H
    ret     0
_f    ENDP
```

Попробуем GCC 4.4.1 без оптимизации:

```
                public f
f               proc near

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], 0FFFFFDFFh
                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`:

```
                public f
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`, эта его часть от 8-го до 15-го бита включительно.

Важное отступление: 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 все еще можно обращаться как к `AX`/`AH`/`AL`.

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`).

Если мы скомпилируем этот же пример не только с включенной оптимизацией `-O3`, но еще и с опцией `regparm=3`, о которой я писал немного выше, то получится еще короче:

```
                public f
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 еще не так хорош для подобных оптимизаций по размеру кода. Впрочем, в реальной жизни, подобные короткие функции лучше всего автоматически делать в виде *inline-функций*[46].

2.14.3 Сдвиги

Битовые сдвиги в Си/Си++ реализованы при помощи операторов `«` и `»`.

Вот этот несложный пример иллюстрирует функцию, считающую количество бит-единиц во входной переменной:

```
#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++;

    return rt;
};
```

В этом цикле, счетчик итераций i считает от 0 до 31, а `1«i` будет от 1 до 0x80000000. Описывая это словами, можно сказать *сдвинуть единицу на n бит влево*. Т.е., в некотором смысле, выражение `1«i` последовательно выдаст все возможные позиции бит в 32-битном числе. Кстати, освободившийся бит справа всегда обнуляется. Макрос `IS_SET` проверяет наличие этого бита в `a`.

Макрос `IS_SET` на самом деле это операция логического И (*AND*) и она возвращает ноль 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 123, поэтому все будет работать.

Компилируем (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:
```

[46] http://en.wikipedia.org/wiki/Inline_function

```
    mov     eax, DWORD PTR _i$[ebp]    ; increment of 1
    add     eax, 1
    mov     DWORD PTR _i$[ebp], eax
$LN4@f:
    cmp     DWORD PTR _i$[ebp], 32     ; 00000020H
    jge     SHORT $LN2@f               ; loop finished?
    mov     edx, 1
    mov     ecx, DWORD PTR _i$[ebp]
    shl     edx, cl                    ; EDX=EDX<<CL
    and     edx, DWORD PTR _a$[ebp]
    je      SHORT $LN1@f               ; result of AND instruction was 0?
                                       ; then skip next instructions
    mov     eax, DWORD PTR _rt$[ebp]   ; no, not zero
    add     eax, 1                     ; increment rt
    mov     DWORD PTR _rt$[ebp], eax
$LN1@f:
    jmp     SHORT $LN3@f
$LN2@f:
    mov     eax, DWORD PTR _rt$[ebp]
    mov     esp, ebp
    pop     ebp
    ret     0
_f    ENDP
```

Вот так работает SHL (*SHift Left*).

Скомпилим то же и в 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]
                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
```

Инструкции сдвига также активно применяются при делении или умножении на числа-степени двойки (1, 2, 4, 8, итд).

Например:

```
unsigned int f(unsigned int a)
{
        return a/4;
};
```

Имеем в итоге (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*) 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.

Для того, чтобы это проще понять, представьте себе десятичную систему счисления и число 23. 23 можно разделить на 10 просто выкинув последний разряд (3, это остаток от деления). После этой операции останется 2 как частное [47].

Так и с умножением. Умножить на 4 это просто сдвинуть число на 2 бита влево, вставив 2 нулевых бита справа (как два самых младших бита).

2.14.4 Пример вычисления CRC32

Это распространенный табличный способ вычисления хеша алгоритмом CRC32[48].

```
/* 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, 0x1adad47d, 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, 0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423,
  0xcfba9599, 0xb8bda50f, 0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
  0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d, 0x76dc4190, 0x01db7106,
  0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
  0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818, 0x7f6a0dbb, 0x086d3d2d,
  0x91646c97, 0xe6635c01, 0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e,
  0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0x12b7e950,
  0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
  0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2, 0x4adfa541, 0x3dd895d7,
  0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0,
  0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9, 0x5005713c, 0x270241aa,
  0xbe0b1010, 0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
  0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17, 0x2eb40d81,
  0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a,
  0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683, 0xe3630b12, 0x94643b84,
```

[47] результат деления

[48] Исходник взят тут: http://burtleburtle.net/bob/c/crc.c

```
  0x0d6d6a3e, 0x7a6a5aa8, 0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
  0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d, 0x806567cb,
  0x196c3671, 0x6e6b06e7, 0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc,
  0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5, 0xd6d6a3e8, 0xa1d1937e,
  0x38d8c2c4, 0x4fdff252, 0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b,
  0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60, 0xdf60efc3, 0xa867df55,
  0x316e8eef, 0x4669be79, 0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
  0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f, 0xc5ba3bbe, 0xb2bd0b28,
  0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d,
  0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a, 0x9c0906a9, 0xeb0e363f,
  0x72076785, 0x05005713, 0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38,
  0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21, 0x86d3d2d4, 0xf1d4e242,
  0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
  0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c, 0x8f659eff, 0xf862ae69,
  0x616bffd3, 0x166ccf45, 0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2,
  0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db, 0xaed16a4a, 0xd9d65adc,
  0x40df0b66, 0x37d83bf0, 0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9,
  0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605, 0xcdd70693,
  0x54de5729, 0x23d967bf, 0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94,
  0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d,
};

/* how to derive the values in crctab[] from polynomial 0xedb88320 */
void build_table()
{
  ub4 i, j;
  for (i=0; i<256; ++i) {
    j = i;
    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);
    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, автор указал 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 will be stored in 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:

; work with bytes using only 32-bit registers. byte from address key+i we store into EDI

    movzx  edi, BYTE PTR [ecx+esi]
    mov    ebx, eax    ; EBX = (hash = len)
    and    ebx, 255    ; EBX = hash & 0xff

; XOR EDI, EBX (EDI=EDI^EBX) - this operation uses all 32 bits of each register
; but other bits (8-31) are cleared all time, so it's OK
; these are cleared because, as for EDI, it was done by MOVZX instruction above
; high bits of EBX was cleared by AND EBX, 255 instruction above (255 = 0xff)

    xor    edi, ebx
    shr    eax, 8                        ; EAX=EAX>>8; bits 24-31 taken "from nowhere"
       will be cleared
    xor    eax, DWORD PTR _crctab[edi*4] ; EAX=EAX^crctab[EDI*4] - choose EDI-th element
       from crctab[] table
    inc    ecx                           ; i++
    cmp    ecx, edx                      ; i<len ?
    jb     SHORT $LL3@crc                ; yes
    pop    edi
    pop    esi
    pop    ebx
$LN1@crc:
    ret    0
_crc    ENDP
```

Попробуем то же самое в GCC 4.4.1 с опцией `-O3`:

```
                public crc
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
                nop                       ; padding
                lea     esi, [esi+0]      ; padding; ESI doesn't changing here

loc_80484B8:
                mov     ecx, eax          ; save previous state of hash to ecx
                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 npad 3.3.

2.15 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.

2.15.1 Пример SYSTEMTIME

Возьмем, к примеру, структуру SYSTEMTIME[49]из win32 описывающую время.

Она объявлена так:

```
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):

```
_t$ = -16           ; size = 16
_main       PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 16      ; 00000010H
    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
    call    _printf
    add     esp, 28       ; 0000001cH
    xor     eax, eax
```

[49]MSDN: SYSTEMTIME structure

```
    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: структура начинается с поля `wYear`. Можно сказать что в качестве аргумента для `GetSystemTime()`[50] передается указатель на структуру SYSTEMTIME, но можно также сказать, что передается указатель на поле `wYear`, что одно и тоже! `GetSystemTime()` пишет текущий год в тот WORD на который указывает переданный указатель, затем сдвигается на 2 байта вправо, пишет текущий месяц, итд, итд.

2.15.2 Выделяем место для структуры через malloc()

Однако, бывает и так, что проще хранить структуры не в стеке а в куче[51]:

```
#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;
};
```

Скомпилируем на этот раз с оптимизацией (`/Ox`) чтобы было проще увидеть то, что нам нужно.

```
_main       PROC
    push    esi
    push    16                      ; 00000010H
    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
```

[50] MSDN: GetSystemTime function
[51] heap

```
    call    _free
    add     esp, 32                         ; 00000020H
    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 eXtent*). Она нужна почти там же где и MOVSX 2.10, 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.

2.15.3 Linux

В Линуксе, для примера, возьем 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:

```
main            proc near
                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 40h
                mov     dword ptr [esp], 0 ; первый агрумент 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
                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 6 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 к `EAX`, но не модифицирует флаги. См. также соответствующий раздел об инструкции `LEA` 3.1.

2.15.4 Упаковка полей в структуре

Достаточно немаловажный момент, это упаковка полей в структурах[52].

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);
};
```

Как видно, мы имеем два поля *char* (занимающий один байт) и еще два — *int* (по 4 байта).

Компилируется это все в:

```
_s$ = 8                 ; size = 16
?f@@YAXUs@@@Z PROC      ; f
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _s$[ebp+12]
    push    eax
    movsx   ecx, BYTE PTR _s$[ebp+8]
    push    ecx
```

[52] См.также: Data structure alignment

```
    mov     edx, DWORD PTR _s$[ebp+4]
    push    edx
    movsx   eax, BYTE PTR _s$[ebp]
    push    eax
    push    OFFSET $SG3842
    call    _printf
    add     esp, 20     ; 00000014H
    pop     ebp
    ret     0
?f@@YAXUs@@@Z ENDP     ; f
_TEXT    ENDS
```

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 кешировать данные из памяти.

Но это не экономично по размеру данных.

Попробуем скомпилировать тот же исходник с опцией (/Zp1) (*/Zp[n] pack structs on n-byte boundary*).

```
_TEXT    SEGMENT
_s$ = 8                ; size = 10
?f@@YAXUs@@@Z PROC     ; f
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _s$[ebp+6]
    push    eax
    movsx   ecx, BYTE PTR _s$[ebp+5]
    push    ecx
    mov     edx, DWORD PTR _s$[ebp+1]
    push    edx
    movsx   eax, BYTE PTR _s$[ebp]
    push    eax
    push    OFFSET $SG3842
    call    _printf
    add     esp, 20     ; 00000014H
    pop     ebp
    ret     0
?f@@YAXUs@@@Z ENDP     ; f
```

Теперь структура занимает 10 байт и все *char* занимают по байту. Что это дает? Экономию места. Недостаток — процессор будет обращаться к этим полям не так эффективно по скорости как мог бы.

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[53]и GCC[54].

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 попадается такое:

```
#include "pshpack1.h"
```

И такое:

```
#include "pshpack4.h"                    // 4 byte packing is the default
```

Сам файл PshPack1.h выглядит так:

```
#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)
```

[53] MSDN: Working with Packing Structures
[54] Structure-Packing Pragmas

```
#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.

2.15.5 Вложенные структуры

Теперь, как насчет ситуаций, когда одна структура определяет внутри себя еще одну структуру?

```
#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)
{
    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);
};
```

... в этом случае, оба поля `inner_struct` просто будут располагаться между полями a,b и d,e в `outer_struct`.

Компилируем (MSVC 2010):

```
_s$ = 8               ; size = 24
_f    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _s$[ebp+20] ; e
    push   eax
    movsx  ecx, BYTE PTR _s$[ebp+16] ; d
    push   ecx
    mov    edx, DWORD PTR _s$[ebp+12] ; c.b
    push   edx
    mov    eax, DWORD PTR _s$[ebp+8] ; c.a
    push   eax
    mov    ecx, DWORD PTR _s$[ebp+4] ; b
    push   ecx
    movsx  edx, BYTE PTR _s$[ebp] ;a
    push   edx
    push   OFFSET $SG2466
    call   _printf
    add    esp, 28    ; 0000001cH
    pop    ebp
    ret    0
_f    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), ситауция будет совсем иная.

2.15.6 Работа с битовыми полями в структуре

Пример CPUID

Язык Си/Си++ позволяет укзывать, 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`[55]. Эта инструкция возвращает информацию о том, какой процессор имеется в наличии и какие фичи он имеет.

Если перед исполнением инструкции в `EAX` будет 1, то `CPUID` вернет упакованную в `EAX` такую информацию о процессоре:

3:0	Stepping
7:4	Model
11:8	Family
13:12	Processor Type
19:16	Extended Model
27:20	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[56].

```
#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
```

[55] http://en.wikipedia.org/wiki/CPUID

[56] Подробнее о встроенном ассемблере GCC

```
#ifdef __GNUC__
    cpuid (1, &b[0], &b[1], &b[2], &b[3]);
#endif

    tmp=(struct CPUID_1_EAX *)&b[0];

    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 из структуры.

Компилируем в MSVC 2008 с опцией `/Ox`:

```
_b$ = -16           ; size = 16
_main    PROC
    sub    esp, 16                      ; 00000010H
    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                      ; 0000000fH
    push   eax
    push   OFFSET $SG15435 ; 'stepping=%d', 0aH, 00H
    call   _printf

    mov    ecx, esi
    shr    ecx, 4
    and    ecx, 15                      ; 0000000fH
    push   ecx
    push   OFFSET $SG15436 ; 'model=%d', 0aH, 00H
    call   _printf

    mov    edx, esi
    shr    edx, 8
    and    edx, 15                      ; 0000000fH
    push   edx
    push   OFFSET $SG15437 ; 'family_id=%d', 0aH, 00H
    call   _printf

    mov    eax, esi
    shr    eax, 12                      ; 0000000cH
    and    eax, 3
    push   eax
    push   OFFSET $SG15438 ; 'processor_type=%d', 0aH, 00H
    call   _printf

    mov    ecx, esi
```

```
    shr     ecx, 16                     ; 00000010H
    and     ecx, 15                     ; 0000000fH
    push    ecx
    push    OFFSET $SG15439 ; 'extended_model_id=%d', 0aH, 00H
    call    _printf

    shr     esi, 20                     ; 00000014H
    and     esi, 255                 ; 000000ffH
    push    esi
    push    OFFSET $SG15440 ; 'extended_family_id=%d', 0aH, 00H
    call    _printf
    add     esp, 48                     ; 00000030H
    pop     esi

    xor     eax, eax
    pop     ebx

    add     esp, 16                     ; 00000010H
    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.

Попробуем GCC 4.4.1 с опцией -O3.

```
main            proc near               ; DATA XREF: _start+17
                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                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
                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 объеденить 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 раннее указывалось в секции о FPU 2.12, и *float* и *double* содержат в себе знак, мантиссу и экспоненту. Однако, можем ли мы работать с этими полями напрямую? Попробуем с *float*.

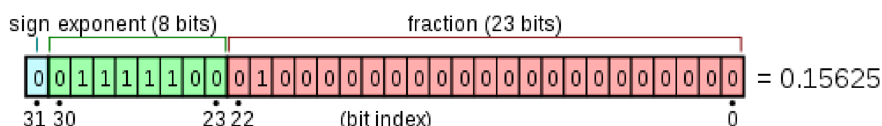

Рис. 2.1: Формат значения float (иллюстрация взята из wikipedia)

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <memory.h>

struct float_as_struct
{
    unsigned int fraction : 23; // fractional part
    unsigned int exponent : 8;  // exponent + 0x3FF
    unsigned int sign : 1;      // sign bit
};

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; // set negative sign
    t.exponent=t.exponent+2; // multiple d by 2^n (n here is 2)

    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 оптимизации:

```
_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          ; 0000000cH

    mov    edx, DWORD PTR _t$[ebp]
    or     edx, -2147483648 ; 80000000H - выставляем знак минус
    mov    DWORD PTR _t$[ebp], edx

    mov    eax, DWORD PTR _t$[ebp]
    shr    eax, 23          ; 00000017H - выкидываем мантиссу
    and    eax, 255         ; 000000ffH - оставляем здесь только экспоненту
    add    eax, 2           ; прибавляем к ней два
    and    eax, 255         ; 000000ffH
    shl    eax, 23          ; 00000017H - пододвигаем результат на место бит 30:23
    mov    ecx, DWORD PTR _t$[ebp]
    and    ecx, -2139095041 ; 807fffffH - выкидываем экспоненту

    ; 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          ; 0000000cH

    fld    DWORD PTR _f$[ebp]

    mov    esp, ebp
    pop    ebp
    ret    0
?f@@YAMM@Z ENDP             ; f
```

Слегка избыточно. В версии скомпиленной 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`?

```
; 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  ; set minus sign
                mov     edx, eax
                and     eax, 807FFFFFh  ; leave only significand and exponent in EAX
                shr     edx, 23         ; prepare exponent
                add     edx, 2          ; add 2
                movzx   edx, dl         ; clear all bits except 7:0 in EAX
                shl     edx, 23         ; shift new calculated exponent to its place
                or      eax, edx        ; add newe exponent and original value without
                    exponent
                mov     [ebp+var_4], eax
                fld     [ebp+var_4]
                leave
                retn
_Z1ff           endp

                public main
main            proc near               ; DATA XREF: _start+17
                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, не взирая на мешанину с полями структуры, GCC умудрился вычислить результат функции `f(1.234)` и сразу подставить его в аргумент для `printf()`!

2.16 Классы в Си++

Я преднамеренно расположил описание классов здесь сразу за структурами, потому что внутреннее представление классов в Си++ почти такое же как и представление структур.

Давайте попробуем простой пример с парой переменных, парой конструкторов и одним методом:

```
#include <stdio.h>

class c
{
private:
    int v1;
    int v2;
public:
    c() // ctor
    {
        v1=667;
        v2=999;
    };

    c(int a, int b) // ctor
    {
        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;
};
```

Вот как выглядит `main()` на ассемблере:

```
_c2$ = -16        ; size = 8
_c1$ = -8         ; size = 8
_main    PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 16               ; 00000010H
    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$[ebp]
    call   ?dump@c@@QAEXXZ      ; c::dump
    lea    ecx, DWORD PTR _c2$[ebp]
    call   ?dump@c@@QAEXXZ      ; 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.

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 3.5.4 — 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*[57].

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 линкер, ни загрузчик DLL операционной системы (мангленные имена могут быть среди экспортов/импортов в DLL), ничего не знают о Си++ или ООП.

Далее вызывается два раза `dump()`.

Теперь смотрим на код в конструкторах:

```
_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       ; 0000029bH
    mov     ecx, DWORD PTR _this$[ebp]
    mov     DWORD PTR [ecx+4], 999     ; 000003e7H
    mov     eax, DWORD PTR _this$[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
    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`, перекладывают его себе в локальную переменную, хотя это и не обязательно.

[57] Wikipedia: Name mangling

И еще метод `dump()`:

```
_this$ = -4             ; size = 4
?dump@c@@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     ; 0000000cH
    mov     esp, ebp
    pop     ebp
    ret     0
?dump@c@@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 будет намного меньше всего:

```
??0c@@QAE@XZ PROC               ; c::c, COMDAT
; _this$ = ecx
    mov     eax, ecx
    mov     DWORD PTR [eax], 667    ; 0000029bH
    mov     DWORD PTR [eax+4], 999  ; 000003e7H
    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@@QAEXXZ 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                 ; 0000000cH
    ret     0
?dump@c@@QAEXXZ 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 у конструктора вместо `RET` имеется `RET 8`.

Это потому что здесь используется thiscall 3.5.4, который, вместе с stdcall 3.5.2 (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 3.5.

Еще, кстати, нужно отметить, что именно компилятор решает, когда вызывать конструктор и деструктор — но это итак известно из основ языка Си++.

2.16.1 GCC

В GCC 4.4.1 все почти так же, за исключением некоторых различий.

```
                public main
main            proc near               ; DATA XREF: _start+17

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
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                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[58]. Во-вторых, указатель на экземпляр передается как первый аргумент функции — конечно же, скрыто от программиста.

Это первый конструктор:

```
                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:

[58] Еще о name mangling разных компиляторов: http://www.agner.org/optimize/calling_conventions.pdf

```
                public _ZN1cC1Eii
_ZN1cC1Eii      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]
                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
```

Эта 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()`:

```
                public _ZN1c4dumpEv
_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 который передается указатель на объект[59] (*this*).

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).

[59] экземпляр класса

2.17 Указатели на функции

Указатель на функцию, в целом, как и любой другой указатель, просто адрес указывающий на начало функции в сегменте кода.

Это применяется часто в т.н. callback-ах [60].

Известные примеры:

- `qsort()`[61], `atexit()`[62] из стандартной библиотеки Си;
- сигналы в *NIX ОС[63];
- запуск тредов: `CreateThread()` (win32), `pthread_create()` (POSIX);
- множество функций win32, например `EnumChildWindows()`[64].

Итак, функция `qsort()` это реализация алгоритма "быстрой сортировки". Функция может сортировать что угодно, любые типы данных, но при условии что вы имеете функцию сравнения двух элементов данных и `qsort()` может вызывать её.

Эта функция сравнения может определяться так:

```
int (*compare)(const void *, const void *)
```

Воспользуемся немного модифицированным примером, который я нашел вот здесь:

```
/* ex3 Sorting ints with qsort */

#include <stdio.h>
#include <stdlib.h>

int comp(const void * _a, const void * _b)
{
  const int *a=(const int *)_a;
  const int *b=(const int *)_b;

  if (*a==*b)
    return 0;
  else
    if (*a < *b)
        return -1;
     else
      return 1;
}

int main(int argc, char* argv[])
{
   int numbers[10]={1892,45,200,-98,4087,5,-12345,1087,88,-100000};
   int i;

  /* Sort the array */
  qsort(numbers,10,sizeof(int),comp) ;
  for (i=0;i<9;i++)
    printf("Number = %d\n",numbers[ i ]) ;
  return 0;
}
```

Компилируем в MSVC 2010 (я убрал некоторые части для краткости) с опцией `/Ox`:

---

[60] http://en.wikipedia.org/wiki/Callback_(computer_science)
[61] http://en.wikipedia.org/wiki/Qsort_(C_standard_library)
[62] http://www.opengroup.org/onlinepubs/009695399/functions/atexit.html
[63] http://en.wikipedia.org/wiki/Signal.h
[64] http://msdn.microsoft.com/en-us/library/ms633494(VS.85).aspx

```
__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$ = -44       ; size = 40
_i$ = -4              ; size = 4
_argc$ = 8            ; size = 4
_argv$ = 12           ; size = 4
_main    PROC
    push    ebp
    mov    ebp, esp
    sub    esp, 44                            ; 0000002cH
    mov    DWORD PTR _numbers$[ebp], 1892       ; 00000764H
    mov    DWORD PTR _numbers$[ebp+4], 45       ; 0000002dH
    mov    DWORD PTR _numbers$[ebp+8], 200      ; 000000c8H
    mov    DWORD PTR _numbers$[ebp+12], -98     ; ffffff9eH
    mov    DWORD PTR _numbers$[ebp+16], 4087    ; 00000ff7H
    mov    DWORD PTR _numbers$[ebp+20], 5
    mov    DWORD PTR _numbers$[ebp+24], -12345  ; ffffcfc7H
    mov    DWORD PTR _numbers$[ebp+28], 1087    ; 0000043fH
    mov    DWORD PTR _numbers$[ebp+32], 88      ; 00000058H
    mov    DWORD PTR _numbers$[ebp+36], -100000 ; fffe7960H
    push    OFFSET _comp
    push    4
    push    10                                ; 0000000aH
    lea    eax, DWORD PTR _numbers$[ebp]
    push    eax
    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()`.

Как `qsort()` вызывает её?

Посмотрим в MSVCR80.DLL (эта DLL куда в MSVC вынесены функции из стандартных библиотек Си):

```
.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
.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 — а проблема это найти ошибку — ведь компилятор на стадии компиляции может вас и не предупредить о потенциальных неприятностях.

2.17.1 GCC

Не слишком большая разница:

```
                lea     eax, [esp+40h+var_28]
                mov     [esp+40h+var_40], eax
                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], 0FFFFCFC7h
                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 враппер для `qsort_r()`.

Она, в свою очередь, вызывает `quicksort()`, где есть вызовы определенной нами функции через переданный указатель:

(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
.text:0002DE00                 mov     [esp+8], edx
.text:0002DE04                 call    [ebp+arg_C]
...
```

2.18 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 также 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.

Когда MMX только появилось, эти регистры на самом деле распологались 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 операционная система 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 операционная система + задача использующая 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, копирование блоков в памяти (`memcpy`), сравнение (`memcmp`), и подобное.

Еще пример: имеется алгоритм шифрования DES, который берет 64-битный блок, 56-битный ключ, шифрует блок с ключем 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[65] — это обработка сразу группы блоков и ключей одновременно. Скажем, на x86 перменная 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*.

Я написал утилиту для перебора паролей/хешей Oracle RDBMS (которые основаны на алгоритме DES), переделав алгоритм bitslice DES для SSE2 и AVX — и теперь возможно шифровать одновременно 128 или 256 блоков-ключей:

http://conus.info/utils/ops_SIMD/

2.18.1 Векторизация

Векторизация[66] 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 кладет в выходной. Важно что операция применяемая ко всем элементам одна и та же. Векторизация — это обрабатывать несколько элементов одновременно.

/IFRUНапример:For example:

```
for (i = 0; i < 1024; i++)
{
    C[i] = A[i]*B[i];
}
```

Этот кусок 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 инстукцию *PMULLD* (*Multiply Packed Signed Dword Integers and Store Low Result*)и `PMULHW` (*Multiply Packed Signed Integers and Store High Result*), можно получить 4 64-битных произведения[67] сразу.

[65] http://www.darkside.com.au/bitslice/

[66] Wikipedia: vectorization

[67] результат умножения

Таким образом, тело цикла исполняется 1024/4 раза вместо 1024, что в 4 раза меньше, и, конечно, быстрее.

Некоторые компиляторы умеют делать автоматическую векторизацию в простых случаях, например Intel C++[68].

Я написал очень простую функцию:

```
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++

Компилирую при помощи Intel C++ 11.1.051 win32:

```
icl intel.cpp /QaxSSE2 /Faintel.asm /Ox
```

Имеем такое (в IDA 6):

```
; int __cdecl f(int, int *, int *, int *)
                public ?f@@YAHHPAH00@Z
?f@@YAHHPAH00@Z proc near

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]
```

[68] Еще о том как Intel C++ умеет автоматически векторизировать циклы: Excerpt: Effective Automatic Vectorization

```
                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 = ar1
                and     edi, 0Fh        ; is ar1 16-byte aligned?
                jz      short loc_9A    ; yes
                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] ; is ar2+i*4 16-byte aligned?
                test    esi, 0Fh
                jz      short loc_109   ; yes!
                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]
                movdqu  xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 is not 16-byte aligned, so
                    load it to xmm0
                paddd   xmm1, xmm0
                movdqa  xmmword ptr [eax+edi*4], xmm1
                add     edi, 4
```

```
                cmp     edi, ecx
                jb      short loc_ED
                jmp     short loc_127
; ---------------------------------------------------------------------------

loc_109:                                ; CODE XREF: f(int,int *,int *,int *)+E3
                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]
                paddd   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
                jmp     short loc_127
?f@@YAHHPAH00@Z endp
```

Инструкции имеющие отношение к SSE2 это:

`MOVDQU` (*Move Unaligned Double Quadword*) — она просто загружает 16 байт из памяти в XMM-

регистр.

PADDD (*Add Packed Integers*) — складывает сразу 4 пары 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 PADDD — адрес значения в памяти, то требуется чтобы адрес был выровнен по 16-байтной границе. Если он не выровнен, произойдет исключение [69].

MOVDQA (*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. MOVDQA работает быстрее чем MOVDQU, но требует вышеозначенного.

Итак, эти SSE2-инструкции исполнятся 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 кусок:

```
                movdqu  xmm0, xmmword ptr [ebx+edi*4] ; ar1+i*4
                paddd   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 is not 16-byte aligned, so
                    load it to xmm0
                paddd   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[70], если компилировать с опциями -O3 и включить поддержку SSE2: -msse2.

Вот что вышло (GCC 4.4.1):

```
; 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
```

[69] О выравнивании данных см. также: Wikipedia: data structure alignment

[70] Подробнее о векторизации в GCC: http://gcc.gnu.org/projects/tree-ssa/vectorization.html

```
                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

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
                paddd   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
                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++.

2.18.2 Реализация strlen() () при помощи SIMD

Прежде всего, следует заметить, что SIMD-инструкции можно вставлять в Си/Си++ код при помощи специальных макросов[71]. В MSVC, часть находится в файле intrin.h.It should be noted that SIMD-instructions may be inserted into Си/Си++ code via special macros[72]. As of MSVC, some of them are located in intrin.h file.

Имеется возможность реализовать функцию strlen()[73] 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.

```
size_t strlen_sse2(const char *str)
{
    register size_t len = 0;
    const char *s=str;
    bool str_is_aligned=(((unsigned int)str)&0xFFFFFFF0) == (unsigned int)str;

    if (str_is_aligned==false)
        return strlen (str);

    __m128i xmm0 = _mm_setzero_si128();
    __m128i xmm1;
    int mask = 0;
```

[71] MSDN: MMX, SSE, and SSE2 Intrinsics
[72] MSDN: MMX, SSE, and SSE2 Intrinsics
[73] strlen() — стандартная функция Си для подсчета длины строки

```
    for (;;)
    {
        xmm1 = _mm_load_si128((__m128i *)s);
        xmm1 = _mm_cmpeq_epi8(xmm1, xmm0);
        if ((mask = _mm_movemask_epi8(xmm1)) != 0)
        {
            unsigned long pos;
            _BitScanForward(&pos, mask);
            len += (size_t)pos;

            break;
        }
        s += sizeof(__m128i);
        len += sizeof(__m128i);
    };

    return len;
}
```

(пример базируется на исходнике отсюда).

Компилируем в MSVC 2010 с опцией `/Ox`:

```
_pos$75552 = -4          ; size = 4
_str$ = 8                ; size = 4
?strlen_sse2@@YAIPBD@Z PROC ; strlen_sse2

    push     ebp
    mov      ebp, esp
    and      esp, -16            ; fffffff0H
    mov      eax, DWORD PTR _str$[ebp]
    sub      esp, 12             ; 0000000cH
    push     esi
    mov      esi, eax
    and      esi, -16            ; fffffff0H
    xor      edx, edx
    mov      ecx, eax
    cmp      esi, eax
    je       SHORT $LN4@strlen_sse
    lea      edx, DWORD PTR [eax+1]
    npad     3
$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
    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
```

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 не взирая 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 ОС 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 присоеденены к реальной физической памяти. Если процесс обратится к блоку памяти, которого не существует, сработает исключение. Так работает виртуальная память[74].

Так вот, функция, читающая сразу по 16 байт, имеет возможность нечаянно вылезти за границу выделенного блока памяти. Предположим, ОС 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., ОС 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:

0x008c1ff8	'h'
0x008c1ff9	'e'
0x008c1ffa	'l'
0x008c1ffb	'l'
0x008c1ffc	'o'
0x008c1ffd	'\x00'
0x008c1ffe	здесь случайный мусор
0x008c1fff	здесь случайный мусор

В обычных условиях, программа вызвает `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, `MOVDQU` 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 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 `pxor xmm0, xmm0` — инструкция просто обнуляет регистр `XMM0`.

`_mm_load_si128()` — это макрос для `MOVDQA`, он просто загружает 16 байт по адресу из указателя в `XMM1`.

[74] http://en.wikipedia.org/wiki/Page_(computer_memory)

`_mm_cmpeq_epi8()` — это макрос для `PCMPEQB`, 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: 11223344556677880000000000000000
XMM0: 11ab3444007877881111111111111111
```

После исполнения `pcmpeqb xmm1, xmm0`, регистр `XMM1` будет содержать:

```
XMM1: ff0000ff0000ffff0000000000000000
```

Эта инструкция в нашем случае, сравнивает каждый 16-байтный блок с блоком состоящим из 16-и нулевых байт, выставленным в `XMM0` при помощи `pxor xmm0, xmm0`.

Следующий макрос `_mm_movemask_epi8()` — это инструкция `PMOVMSKB`.

Она очень удобна как раз для использования в паре с `PCMPEQB`.

`pmovmskb eax, xmm1`

Эта инструкция выставит самый первый бит `EAX` в еденицу, 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` являются 0xff?* 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 `hello\x00garbage\x00ab`

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 (я использую здесь порядок с MSB[75]до LSB[76]):

```
XMM1: 0000ff00000000000000ff0000000000
```

Это означает что инструкция сравнения обнаружила два нулевых байта, что и не удивительно.

`PMOVMSKB` в нашем случае подготовит `EAX` вот так (в двоичном представлении): *0010000000100000b*.

Совершенно очевидно что далее наша функция должна учитывать только первый встретившийся ноль 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=0010000000100000b
```

После исполнения этой инструкции `bsf 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

[75]most significant bit
[76]least significant bit

2.19 x86-64

Это расширение x86-архитуктуры 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 регистры общего пользования: `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`.

  У новых регистров `r8-r15` также имеются их *младшие части*: `r8d-r15d` (младшие 32-битные части), `r8w-r15w` (младшие 16-битные части), `r8b-r15b` (младшие 8-битные части).

  Удвоено количество SIMD-регистров: с 8 до 16: `XMM0-XMM15`.

- В win64 передача всех параметров немного иная, это немного похоже на fastcall 3.5.3. 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 бóльшие функции могут сохранять их значения на будущее.

  См.также в соответствующем разделе о способах передачи аргументов через стек 3.5.

- Сишный *int* 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*[77]. Для нас это означает, что в итоговом коде будет меньше локальных переменных.

Для примера, функция вычисляющая первый S-бокс 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 2.18):

```
/*
 * 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
```

[77] распределение переменных по регистрам

```
) {
    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;
    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;
    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`:

```
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
    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$[esp+32], edi
    xor     edi, ecx
    not     edi
    and     edi, edx
    mov     ebx, edi
    and     ebx, ebp
    xor     ebx, DWORD PTR _x28$[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
    pop     ebp
    mov     DWORD PTR [ecx], eax
    pop     ebx
    add     esp, 20                     ; 00000014H
    ret     0
_s1     ENDP
```

5 переменных компилятору пришлось разместить в локальном стеке.

Теперь попробуем то же самое только в 64-битной версии 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
    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$[rsp]
    xor     rbx, rax
    mov     rax, QWORD PTR out4$[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
s1    ENDP
```

Компилятор ничего не выделил в локальном стеке, а `x36` это синоним для `a5`.

Кстати, видно что функция сохраняет регистры `RCX`, `RDX` в отведенных для этого вызываемой функцией местах, а `R8` и `R9` не сохраняет, а начинает использовать их сразу.

Кстати, существуют процессоры с еще большим количеством регистров общего использования, например, Itanium — 128 регистров.

Глава 3

Еще кое-что

3.1 Инструкция LEA

`LEA` (*Load Effective Address*) это инструкция которая задумывалась вовсе не для складывания чисел, а для формирования адреса например из указателя на массив и прибавления индекса к нему[1].

Важная особенность `LEA` в том что производимые ею вычисления не модифицируют флаги.

```
int f(int a, int b)
{
        return a*8+b;
};
```

Компилируем в MSVC 2010 с `/Ox`:

```
_a$ = 8                                                         ; size = 4
_b$ = 12                                                        ; size = 4
_f      PROC
        mov     eax, DWORD PTR _b$[esp-4]
        mov     ecx, DWORD PTR _a$[esp-4]
        lea     eax, DWORD PTR [eax+ecx*8]
        ret     0
_f      ENDP
```

[1] См. также: http://en.wikipedia.org/wiki/Addressing_mode

3.2 Пролог и эпилог в функции

Пролог функции это инструкции в самом начале функции. Как правило это что-то вроде:

```
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 анулирует выделенное место в стеке, возвращает значение EBP на то что было и возвращает управление в вызывающую функцию:

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

Наличие эпилога и пролога может несколько ухудшить эффективность рекурсии.

Например, однажды я написал функцию для поиска нужного узла дерева. 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 есть такая вещь как хвостовая рекурсия[2]: когда компилятор или интерпретатор превращает рекурсию (с которой возможно это проделать: *хвостовую*) в итерацию для эффективности.

[2] http://en.wikipedia.org/wiki/Tail_call

3.3 npad

Это макрос в ассемблере, для выравнивания некоторой метки по некоторой границе.

Это нужно для тех *нагруженных* меток, куда чаще всего передается управление, например, начало цикла. Для того чтобы процессор мог эффективнее вытягивать данные или код из памяти, через шину с памятью, кеширование, итд.

Взято из `listing.inc` (MSVC):

Это, кстати, любопытный пример различных вариантов `NOP`-ов. Все эти инструкции не дают никакого эффекта, но отличаются разной длиной.

```
;; LISTING.INC
;;
;; This file contains assembler macros and is included by the files created
;; with the -FA compiler switch to be assembled by MASM (Microsoft Macro
;; Assembler).
;;
;; Copyright (c) 1993-2003, Microsoft Corporation. All rights reserved.

;; non destructive nops
npad macro size
if size eq 1
  nop
else
 if size eq 2
   mov edi, edi
 else
  if size eq 3
    ; lea ecx, [ecx+00]
    DB 8DH, 49H, 00H
  else
   if size eq 4
     ; lea esp, [esp+00]
     DB 8DH, 64H, 24H, 00H
   else
    if size eq 5
      add eax, DWORD PTR 0
    else
     if size eq 6
       ; lea ebx, [ebx+00000000]
       DB 8DH, 9BH, 00H, 00H, 00H, 00H
     else
      if size eq 7
        ; lea esp, [esp+00000000]
        DB 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
      else
       if size eq 8
        ; jmp .+8; .npad 6
        DB 0EBH, 06H, 8DH, 9BH, 00H, 00H, 00H, 00H
       else
        if size eq 9
         ; jmp .+9; .npad 7
         DB 0EBH, 07H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
        else
         if size eq 10
          ; jmp .+A; .npad 7; .npad 1
          DB 0EBH, 08H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 90H
         else
          if size eq 11
           ; jmp .+B; .npad 7; .npad 2
           DB 0EBH, 09H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8BH, 0FFH
          else
           if size eq 12
            ; jmp .+C; .npad 7; .npad 3
            DB 0EBH, 0AH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 49H, 00H
           else
```

```
            if size eq 13
             ; jmp .+D; .npad 7; .npad 4
             DB 0EBH, 0BH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 64H, 24H, 00H
            else
             if size eq 14
              ; jmp .+E; .npad 7; .npad 5
              DB 0EBH, 0CH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 05H, 00H, 00H, 00H, 00H
             else
              if size eq 15
               ; jmp .+F; .npad 7; .npad 6
               DB 0EBH, 0DH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 9BH, 00H, 00H, 00H,
                   00H
              else
               %out error: unsupported npad size
               .err
              endif
             endif
            endif
           endif
          endif
         endif
        endif
       endif
      endif
     endif
    endif
   endif
  endif
 endif
endif
endm
```

3.4 Представление знака в числах

Методов представления чисел с знаком "плюс"или "минус"несколько[3], а в x86 применяется метод "дополнительный код"или "two's complement".

Разница в подходе к знаковым/беззнаковым числам, собственно, нужна потому что, например, если представить 0xFFFFFFFE и 0x0000002 как беззнаковое, то первое число (4294967294) больше второго (2). Если их оба представить как знаковые, то первое будет -2, которое, разумеется, меньше чем второе (2). Вот почему инструкции для условных переходов 2.7 представлены в обоих версиях — и для знаковых сравнений (например `JG`, `JL`) и для беззнаковых (`JA`, `JBE`).

3.4.1 Переполнение integer

Бывает так, что ошибки представления знаковых/беззнаковых могут привести к уязвимости *переполнение integer*.

Например, есть некий сервис, который принимает по сети некие пакеты. В пакете есть заголовок где указана длина пакета. Это 32-битное значение. В процессе приема пакета, сервис проверяет это значение и сверяет, больше ли оно чем максимальный размер пакета, скажем, константа `MAX_PACKET_SIZE` (например, 10 килобайт). Сравнение знаковое. Злоумышленник подставляет значение 0xFFFFFFFF. Это число трактуется как знаковое -1 и оно меньше чем 10000. Проверка проходит. Продолжаем дальше и копируем этот пакет куда-нибудь себе в сегмент данных... вызов функции `memcpy (dst, src, 0xFFFFFFFF)` скорее всего, затрет много чего внутри процесса.

Немного подробнее: http://www.phrack.org/issues.html?issue=60&id=10

[3] http://en.wikipedia.org/wiki/Signed_number_representations

3.5 Способы передачи аргументов при вызове функций

3.5.1 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 ESP в нормальное состояние, после возврата вызываемой функции.

```
push arg3
push arg2
push arg3
call function
add esp, 12 ; return ESP
```

3.5.2 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)`[4]. Вызывающая функция не будет корректировать указатель стека при помощи инструкции `add esp, x`.

```
push arg3
push arg2
push arg1
call function

function:
... do something ...
ret 12
```

Этот способ используется почти везде в системных библиотеках win32, но не в win64 (о win64 смотрите ниже).

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 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 скомпилена 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*.

3.5.3 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*. Это не стандартизированый 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, возможно будут проблемы.

MSVC и GCC передает первый и второй аргумент через ECX и EDX а остальные аргументы через стек. Вызываемая функция возвращает указатель стека в первоначальное состояние.

[4] Размер переменной типа *int* — 4 в x86-системах и 8 в x64-системах

Указатель стека должен быть возвращен в первоначальное состояние вызываемой функцией, как в случае *stdcall*.

```
push arg3
mov edx, arg2
mov ecx, arg1
call function

function:
.. do something ..
ret 4
```

GCC regparm

Это в некотором роде, развитие *fastcall*[5]. 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.

3.5.4 thiscall

В C++, это передача в функцию-метод указателя *this* на объект.

В MSVC указатель *this* обычно передается в регистре `ECX`.

В GCC указатель *this* обычно передается как самый первый аргумент. Таким образом, внутри будет видно, что у всех функций-методов на один аргумент больше.

3.5.5 x86-64

win64

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 бóльшие 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).

3.5.6 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 регистре `XMM0` вместо `ST(0)`.

[5] http://www.ohse.de/uwe/articles/gcc-attributes.html#func-regparm

Глава 4

Поиск в коде того что нужно

Современный софт, в общем-то, минимализмом не отличается.

Но не потому, что программисты слишком много написали, а потому что к исполняемым файлам обыкновенно прикомпиливают все подряд библиотеки. Если бы все вспомогательные библиотеки всегда выносили во внешние DLL, мир был бы иным.

Таким образом, очень полезно сразу понимать, какая функция из стандартной библиотеки или более-менее известной (как Boost[1], libpng[2]), 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 задач reverse engineer-а это быстрый поиск в коде того что собственно его интересует.

Дизассемблер IDA 6 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`, итд.

Когда вы пытаетесь понять, что делает тот или иной код, это запросто может быть какая-то опенсорсная библиотека вроде libpng. Поэтому когда находите константы, или текстовые строки которые выглядят явно знакомыми, всегда полезно их погуглить. А если вы найдете искомый опенсорсный проект где это используется, то тогда будет достаточно будет просто сравнить вашу функцию с ней. Это решит часть проблем.

К примеру, однажды я 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. Я в конце концов пришел к тому что одна из функций декомпрессирующая пакеты называется CsDecomprLZC(). Не сильно раздумывая, я решил погуглить и оказалось что функция с таким же названием имеется в MaxDB (это опен-сорсный проект SAP).

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.

4.1 Связь с внешним миром

Первое на что нужно обратить внимание, это какие функции из API операционной системы 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 об 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

[1] http://www.boost.org/

[2] http://www.libpng.org/pub/png/libpng.html

мест эта функция вызывается и что самое главное: как используется результат этой функции.

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): http://blogs.conus.info/node/44.

4.2 Строки

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 дампятся значения некоторых локальных или глобальных переменных, это тоже может помочь, как минимум, узнать их имена. Например, в Oracle RDBMS одна из таких функций: `ksdwrt()`.

Может также помочь наличие `assert()` в коде: обычно этот макрос оставляет название файла-исходника, номер строки, и условие.

Осмысленные текстовые строки вообще очень сильно могут помочь. Дизассемблер IDA 6 может сразу указать, из какой функции и из какого её места используется эта строка. Попадаются и смешные случаи.

Парадоксально, но сообщения об ошибках также могут помочь найти то что нужно. В Oracle RDBMS сигнализация об ошибках проходит при помощи вызова некоторой группы функций. Тут еще немного об этом.

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.

4.3 Константы

Некоторые алгоритмы, особенно криптографические, используют хорошо различимые константы, которые при помощи IDA 6 легко находить в коде.

Например алгоритм MD5[3] 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.

4.3.1 Magic numbers

Немало форматов файлов определяет стандартный заголовок файла где используются *magic numbers*[4].

Скажем, все исполняемые файлы для Win32 и MS-DOS начинаются с двух символов "MZ"[5].

В начале MIDI-файла должно быть "MThd". Если у нас есть использующая для чего-нибудь MIDI-файлы программа очень вероятно, что 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)

[3] http://ru.wikipedia.org/wiki/MD5

[4] http://en.wikipedia.org/wiki/Magic_number_(programming)

[5] http://en.wikipedia.org/wiki/DOS_MZ_executable

```
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`.

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.

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. *Что-либо* что получает пакеты по DHCP должно 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 `DhcpExtractOptionsForVali` и `DhcpExtractFullOptions()`:

```
.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:

```
.text:000007FF6480875F                  mov     eax, [rsi]
.text:000007FF64808761                  cmp     eax, cs:dword_7FF6483CBE8
.text:000007FF64808767                  jnz     loc_7FF64817179
```

И:

```
.text:000007FF648082C7                  mov     eax, [r12]
.text:000007FF648082CB                  cmp     eax, cs:dword_7FF6483CBEC
.text:000007FF648082D1                  jnz     loc_7FF648173AF
```

4.4 Поиск нужных инструкций

Если программа использует инструкции сопроцессора, и их не очень много, то можно попробовать 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 6, 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 отладчике или tracer 6.0.1 (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
FPU StatusWord=
FPU ST(0): 1.000000
```

В `ST(0)` содержится первый аргумент (1), второй содержится в `[ebx]`.

Следующая за `FDIV` инструкция 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[6], модифицировать его на лету:

```
gt -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.

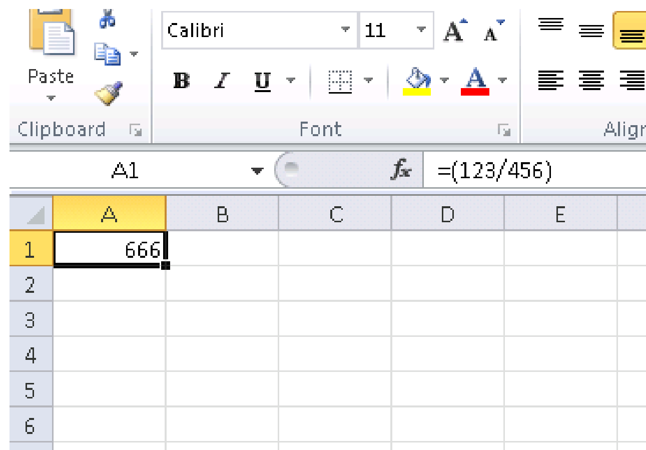

Рис. 4.1: Пранк сработал

Если попробовать ту же версию Excel, только x64, то окажется что там инструкций `FDIV` всего 12, причем нужная нам — третья по счету.

```
gt64.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).

[6]practical joke

4.5 Подозрительные паттерны кода

Современные компиляторы не генерируют инструкции `LOOP` и `RCL`. С другой стороны, эти инструкции хорошо знакомы кодерам предпочитающим писать прямо на ассемблере. Если такие инструкции встретились, можно сказать с какой-то вероятностью, что этот кусок кода написан вручную.

Глава 5

Задачи

Почти для всех задач, если не указано иное, два вопроса:

1) Что делает эта функция? Ответ должен состоять из одной фразы.

2) Перепишите эту функцию на Си/Си++.

Подсказки и ответы собраны в приложении к этой брошюре.

5.1 Легкий уровень

5.1.1 Задача 1.1

Это стандартная функция из библиотек Си. Исходник взят из OpenWatcom. Скомпилировано в MSVC 2010.

```
_TEXT    SEGMENT
_input$ = 8                        ; size = 1
_f PROC
    push   ebp
    mov    ebp, esp
    movsx  eax, BYTE PTR _input$[ebp]
    cmp    eax, 97           ; 00000061H
    jl     SHORT $LN1@f
    movsx  ecx, BYTE PTR _input$[ebp]
    cmp    ecx, 122          ; 0000007aH
    jg     SHORT $LN1@f
    movsx  edx, BYTE PTR _input$[ebp]
    sub    edx, 32           ; 00000020H
    mov    BYTE PTR _input$[ebp], dl
$LN1@f:
    mov    al, BYTE PTR _input$[ebp]
    pop    ebp
    ret    0
_f ENDP
_TEXT    ENDS
```

Это он же скомпилирован при помощи GCC 4.4.1 с опцией `-O3` (максимальная оптимизация):

```
_f              proc near

input           = dword ptr  8

                push    ebp
                mov     ebp, esp
                movzx   eax, byte ptr [ebp+input]
                lea     edx, [eax-61h]
                cmp     dl, 19h
                ja      short loc_80483F2
                sub     eax, 20h

loc_80483F2:
```

```
                pop     ebp
                retn
_f              endp
```

5.1.2 Задача 1.2

Это также стандартная функция из библиотек Си. Исходник взят из OpenWatcom и немного переделан. Скомпилировано в MSVC 2010 с флагом (`/Ox`).

Эта функция использует стандартные функции Си: isspace() и isdigit().

```
EXTRN    _isdigit:PROC
EXTRN    _isspace:PROC
EXTRN    ___ptr_check:PROC
; Function compile flags: /Ogtpy
_TEXT    SEGMENT
_p$ = 8                             ; size = 4
_f    PROC
    push   ebx
    push   esi
    mov    esi, DWORD PTR _p$[esp+4]
    push   edi
    push   0
    push   esi
    call   ___ptr_check
    mov    eax, DWORD PTR [esi]
    push   eax
    call   _isspace
    add    esp, 12                      ; 0000000cH
    test   eax, eax
    je     SHORT $LN6@f
    npad   2
$LL7@f:
    mov    ecx, DWORD PTR [esi+4]
    add    esi, 4
    push   ecx
    call   _isspace
    add    esp, 4
    test   eax, eax
    jne    SHORT $LL7@f
$LN6@f:
    mov    bl, BYTE PTR [esi]
    cmp    bl, 43                     ; 0000002bH
    je     SHORT $LN4@f
    cmp    bl, 45                     ; 0000002dH
    jne    SHORT $LN5@f
$LN4@f:
    add    esi, 4
$LN5@f:
    mov    edx, DWORD PTR [esi]
    push   edx
    xor    edi, edi
    call   _isdigit
    add    esp, 4
    test   eax, eax
    je     SHORT $LN2@f
$LL3@f:
    mov    ecx, DWORD PTR [esi]
    mov    edx, DWORD PTR [esi+4]
    add    esi, 4
    lea    eax, DWORD PTR [edi+edi*4]
    push   edx
    lea    edi, DWORD PTR [ecx+eax*2-48]
    call   _isdigit
    add    esp, 4
    test   eax, eax
```

```
    jne     SHORT $LL3@f
$LN2@f:
    cmp     bl, 45                      ; 0000002dH
    jne     SHORT $LN14@f
    neg     edi
$LN14@f:
    mov     eax, edi
    pop     edi
    pop     esi
    pop     ebx
    ret     0
_f    ENDP
_TEXT    ENDS
```

То же скомпилировано в GCC 4.4.1. Задача немного усложняется тем, что GCC представил isspace() и isdigit() как inline-функции и вставил их тела прямо в код.

```
_f              proc near

var_10          = dword ptr -10h
var_9           = byte ptr -9
input           = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 18h
                jmp     short loc_8048410
loc_804840C:
                add     [ebp+input], 4

loc_8048410:
                call    ___ctype_b_loc
                mov     edx, [eax]
                mov     eax, [ebp+input]
                mov     eax, [eax]
                add     eax, eax
                lea     eax, [edx+eax]
                movzx   eax, word ptr [eax]
                movzx   eax, ax
                and     eax, 2000h
                test    eax, eax
                jnz     short loc_804840C
                mov     eax, [ebp+input]
                mov     eax, [eax]
                mov     [ebp+var_9], al
                cmp     [ebp+var_9], '+'
                jz      short loc_8048444
                cmp     [ebp+var_9], '-'
                jnz     short loc_8048448

loc_8048444:
                add     [ebp+input], 4

loc_8048448:
                mov     [ebp+var_10], 0
                jmp     short loc_8048471

loc_8048451:
                mov     edx, [ebp+var_10]
                mov     eax, edx
                shl     eax, 2
                add     eax, edx
                add     eax, eax
                mov     edx, eax
                mov     eax, [ebp+input]
                mov     eax, [eax]
```

```
                lea     eax, [edx+eax]
                sub     eax, 30h
                mov     [ebp+var_10], eax
                add     [ebp+input], 4

loc_8048471:
                call    ___ctype_b_loc
                mov     edx, [eax]
                mov     eax, [ebp+input]
                mov     eax, [eax]
                add     eax, eax
                lea     eax, [edx+eax]
                movzx   eax, word ptr [eax]
                movzx   eax, ax
                and     eax, 800h
                test    eax, eax
                jnz     short loc_8048451
                cmp     [ebp+var_9], 2Dh
                jnz     short loc_804849A
                neg     [ebp+var_10]

loc_804849A:
                mov     eax, [ebp+var_10]
                leave
                retn
_f              endp
```

5.1.3 Задача 1.3

Это также стандартная функция из библиотек Си, а вернее, две функции, работающие в паре. Исходник взят из MSVC 2010 и немного переделан.

Суть переделки в том, что эта функция может корректно работать в мульти-тредовой среде, а я, для упрощения (или запутывания) убрал поддержку этого.

Скомпилировано в MSVC 2010 с флагом (`/Ox`).

```
_BSS    SEGMENT
_v    DD    01H DUP (?)
_BSS    ENDS

_TEXT    SEGMENT
_s$ = 8                             ; size = 4
f1    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _s$[ebp]
    mov    DWORD PTR _v, eax
    pop    ebp
    ret    0
f1    ENDP
_TEXT    ENDS
PUBLIC    f2

_TEXT    SEGMENT
f2    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _v
    imul   eax, 214013      ; 000343fdH
    add    eax, 2531011     ; 00269ec3H
    mov    DWORD PTR _v, eax
    mov    eax, DWORD PTR _v
    shr    eax, 16          ; 00000010H
    and    eax, 32767       ; 00007fffH
    pop    ebp
```

```
    ret     0
f2    ENDP
_TEXT    ENDS
END
```

То же скомпилировано при помощи GCC 4.4.1:

```
                public f1
f1              proc near

arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0]
                mov     ds:v, eax
                pop     ebp
                retn
f1              endp

                public f2
f2              proc near
                push    ebp
                mov     ebp, esp
                mov     eax, ds:v
                imul    eax, 343FDh
                add     eax, 269EC3h
                mov     ds:v, eax
                mov     eax, ds:v
                shr     eax, 10h
                and     eax, 7FFFh
                pop     ebp
                retn
f2              endp

bss             segment dword public 'BSS' use32
                assume cs:_bss
                dd ?
bss             ends
```

5.1.4 Задача 1.4

Это стандартная функция из библиотек Си. Исходник взят из MSVC 2010. Скомпилировано в MSVC 2010 с флагом `/Ox`.

```
PUBLIC    _f
_TEXT    SEGMENT
_arg1$ = 8            ; size = 4
_arg2$ = 12           ; size = 4
_f    PROC
    push   esi
    mov    esi, DWORD PTR _arg1$[esp]
    push   edi
    mov    edi, DWORD PTR _arg2$[esp+4]
    cmp    BYTE PTR [edi], 0
    mov    eax, esi
    je     SHORT $LN7@f
    mov    dl, BYTE PTR [esi]
    push   ebx
    test   dl, dl
    je     SHORT $LN4@f
    sub    esi, edi
    npad   6
$LL5@f:
    mov    ecx, edi
```

```
    test   dl, dl
    je     SHORT $LN2@f
$LL3@f:
    mov    dl, BYTE PTR [ecx]
    test   dl, dl
    je     SHORT $LN14@f
    movsx  ebx, BYTE PTR [esi+ecx]
    movsx  edx, dl
    sub    ebx, edx
    jne    SHORT $LN2@f
    inc    ecx
    cmp    BYTE PTR [esi+ecx], bl
    jne    SHORT $LL3@f
$LN2@f:
    cmp    BYTE PTR [ecx], 0
    je     SHORT $LN14@f
    mov    dl, BYTE PTR [eax+1]
    inc    eax
    inc    esi
    test   dl, dl
    jne    SHORT $LL5@f
    xor    eax, eax
    pop    ebx
    pop    edi
    pop    esi
    ret    0
_f    ENDP
_TEXT    ENDS
END
```

То же скомпилировано при помощи GCC 4.4.1:

```
                public f
f               proc near

var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4
arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
                sub     esp, 10h
                mov     eax, [ebp+arg_0]
                mov     [ebp+var_4], eax
                mov     eax, [ebp+arg_4]
                movzx   eax, byte ptr [eax]
                test    al, al
                jnz     short loc_8048443
                mov     eax, [ebp+arg_0]
                jmp     short locret_8048453

loc_80483F4:
                mov     eax, [ebp+var_4]
                mov     [ebp+var_8], eax
                mov     eax, [ebp+arg_4]
                mov     [ebp+var_C], eax
                jmp     short loc_804840A

loc_8048402:
                add     [ebp+var_8], 1
                add     [ebp+var_C], 1

loc_804840A:
                mov     eax, [ebp+var_8]
                movzx   eax, byte ptr [eax]
```

```
                test    al, al
                jz      short loc_804842E
                mov     eax, [ebp+var_C]
                movzx   eax, byte ptr [eax]
                test    al, al
                jz      short loc_804842E
                mov     eax, [ebp+var_8]
                movzx   edx, byte ptr [eax]
                mov     eax, [ebp+var_C]
                movzx   eax, byte ptr [eax]
                cmp     dl, al
                jz      short loc_8048402

loc_804842E:
                mov     eax, [ebp+var_C]
                movzx   eax, byte ptr [eax]
                test    al, al
                jnz     short loc_804843D
                mov     eax, [ebp+var_4]
                jmp     short locret_8048453

loc_804843D:
                add     [ebp+var_4], 1
                jmp     short loc_8048444


loc_8048443:
                nop

loc_8048444:
                mov     eax, [ebp+var_4]
                movzx   eax, byte ptr [eax]
                test    al, al
                jnz     short loc_80483F4
                mov     eax, 0

locret_8048453:
                leave
                retn
f               endp
```

5.1.5 Задача 1.5

Задача, скорее, на эрудицию, нежели на чтение кода.

Функция взята из OpenWatcom. Скомпилировано в MSVC 2010 с флагом `/Ox`.

```
_DATA    SEGMENT
COMM     __v:DWORD
_DATA    ENDS
PUBLIC   __real@3e45798ee2308c3a
PUBLIC   __real@4147ffff80000000
PUBLIC   __real@4150017ec0000000
PUBLIC   _f
EXTRN    __fltused:DWORD
CONST    SEGMENT
__real@3e45798ee2308c3a DQ 03e45798ee2308c3ar    ; 1e-008
__real@4147ffff80000000 DQ 04147ffff80000000r    ; 3.14573e+006
__real@4150017ec0000000 DQ 04150017ec0000000r    ; 4.19584e+006
CONST    ENDS
_TEXT    SEGMENT
_v1$ = -16              ; size = 8
_v2$ = -8               ; size = 8
_f    PROC
    sub    esp, 16      ; 00000010H
    fld    QWORD PTR __real@4150017ec0000000
```

```
    fstp    QWORD PTR _v1$[esp+16]
    fld     QWORD PTR __real@4147ffff80000000
    fstp    QWORD PTR _v2$[esp+16]
    fld     QWORD PTR _v1$[esp+16]
    fld     QWORD PTR _v1$[esp+16]
    fdiv    QWORD PTR _v2$[esp+16]
    fmul    QWORD PTR _v2$[esp+16]
    fsubp   ST(1), ST(0)
    fcomp   QWORD PTR __real@3e45798ee2308c3a
    fnstsw  ax
    test    ah, 65         ; 00000041H
    jne     SHORT $LN1@f
    or      DWORD PTR __v, 1
$LN1@f:
    add     esp, 16        ; 00000010H
    ret     0
_f    ENDP
_TEXT    ENDS
```

5.1.6 Задача 1.6

Скомпилировано в MSVC 2010 с ключом `/Ox`.

```
PUBLIC    _f
; Function compile flags: /Ogtpy
_TEXT    SEGMENT
_k0$ = -12            ; size = 4
_k3$ = -8             ; size = 4
_k2$ = -4             ; size = 4
_v$ = 8               ; size = 4
_k1$ = 12             ; size = 4
_k$ = 12              ; size = 4
_f    PROC
    sub     esp, 12    ; 0000000cH
    mov     ecx, DWORD PTR _v$[esp+8]
    mov     eax, DWORD PTR [ecx]
    mov     ecx, DWORD PTR [ecx+4]
    push    ebx
    push    esi
    mov     esi, DWORD PTR _k$[esp+16]
    push    edi
    mov     edi, DWORD PTR [esi]
    mov     DWORD PTR _k0$[esp+24], edi
    mov     edi, DWORD PTR [esi+4]
    mov     DWORD PTR _k1$[esp+20], edi
    mov     edi, DWORD PTR [esi+8]
    mov     esi, DWORD PTR [esi+12]
    xor     edx, edx
    mov     DWORD PTR _k2$[esp+24], edi
    mov     DWORD PTR _k3$[esp+24], esi
    lea     edi, DWORD PTR [edx+32]
$LL8@f:
    mov     esi, ecx
    shr     esi, 5
    add     esi, DWORD PTR _k1$[esp+20]
    mov     ebx, ecx
    shl     ebx, 4
    add     ebx, DWORD PTR _k0$[esp+24]
    sub     edx, 1640531527   ; 61c88647H
    xor     esi, ebx
    lea     ebx, DWORD PTR [edx+ecx]
    xor     esi, ebx
    add     eax, esi
    mov     esi, eax
    shr     esi, 5
```

```
    add     esi, DWORD PTR _k3$[esp+24]
    mov     ebx, eax
    shl     ebx, 4
    add     ebx, DWORD PTR _k2$[esp+24]
    xor     esi, ebx
    lea     ebx, DWORD PTR [edx+eax]
    xor     esi, ebx
    add     ecx, esi
    dec     edi
    jne     SHORT $LL8@f
    mov     edx, DWORD PTR _v$[esp+20]
    pop     edi
    pop     esi
    mov     DWORD PTR [edx], eax
    mov     DWORD PTR [edx+4], ecx
    pop     ebx
    add     esp, 12                     ; 0000000cH
    ret     0
_f    ENDP
```

5.1.7 Задача 1.7

Это взята функция из ядра Linux 2.6.

Скомпилировано в MSVC 2010 с опцией `/Ox`:

```
_table    db 000h, 080h, 040h, 0c0h, 020h, 0a0h, 060h, 0e0h
    db 010h, 090h, 050h, 0d0h, 030h, 0b0h, 070h, 0f0h
    db 008h, 088h, 048h, 0c8h, 028h, 0a8h, 068h, 0e8h
    db 018h, 098h, 058h, 0d8h, 038h, 0b8h, 078h, 0f8h
    db 004h, 084h, 044h, 0c4h, 024h, 0a4h, 064h, 0e4h
    db 014h, 094h, 054h, 0d4h, 034h, 0b4h, 074h, 0f4h
    db 00ch, 08ch, 04ch, 0cch, 02ch, 0ach, 06ch, 0ech
    db 01ch, 09ch, 05ch, 0dch, 03ch, 0bch, 07ch, 0fch
    db 002h, 082h, 042h, 0c2h, 022h, 0a2h, 062h, 0e2h
    db 012h, 092h, 052h, 0d2h, 032h, 0b2h, 072h, 0f2h
    db 00ah, 08ah, 04ah, 0cah, 02ah, 0aah, 06ah, 0eah
    db 01ah, 09ah, 05ah, 0dah, 03ah, 0bah, 07ah, 0fah
    db 006h, 086h, 046h, 0c6h, 026h, 0a6h, 066h, 0e6h
    db 016h, 096h, 056h, 0d6h, 036h, 0b6h, 076h, 0f6h
    db 00eh, 08eh, 04eh, 0ceh, 02eh, 0aeh, 06eh, 0eeh
    db 01eh, 09eh, 05eh, 0deh, 03eh, 0beh, 07eh, 0feh
    db 001h, 081h, 041h, 0c1h, 021h, 0a1h, 061h, 0e1h
    db 011h, 091h, 051h, 0d1h, 031h, 0b1h, 071h, 0f1h
    db 009h, 089h, 049h, 0c9h, 029h, 0a9h, 069h, 0e9h
    db 019h, 099h, 059h, 0d9h, 039h, 0b9h, 079h, 0f9h
    db 005h, 085h, 045h, 0c5h, 025h, 0a5h, 065h, 0e5h
    db 015h, 095h, 055h, 0d5h, 035h, 0b5h, 075h, 0f5h
    db 00dh, 08dh, 04dh, 0cdh, 02dh, 0adh, 06dh, 0edh
    db 01dh, 09dh, 05dh, 0ddh, 03dh, 0bdh, 07dh, 0fdh
    db 003h, 083h, 043h, 0c3h, 023h, 0a3h, 063h, 0e3h
    db 013h, 093h, 053h, 0d3h, 033h, 0b3h, 073h, 0f3h
    db 00bh, 08bh, 04bh, 0cbh, 02bh, 0abh, 06bh, 0ebh
    db 01bh, 09bh, 05bh, 0dbh, 03bh, 0bbh, 07bh, 0fbh
    db 007h, 087h, 047h, 0c7h, 027h, 0a7h, 067h, 0e7h
    db 017h, 097h, 057h, 0d7h, 037h, 0b7h, 077h, 0f7h
    db 00fh, 08fh, 04fh, 0cfh, 02fh, 0afh, 06fh, 0efh
    db 01fh, 09fh, 05fh, 0dfh, 03fh, 0bfh, 07fh, 0ffh

f               proc near

arg_0           = dword ptr  4

                mov     edx, [esp+arg_0]
                movzx   eax, dl
                movzx   eax, _table[eax]
```

```
                mov     ecx, edx
                shr     edx, 8
                movzx   edx, dl
                movzx   edx, _table[edx]
                shl     ax, 8
                movzx   eax, ax
                or      eax, edx
                shr     ecx, 10h
                movzx   edx, cl
                movzx   edx, _table[edx]
                shr     ecx, 8
                movzx   ecx, cl
                movzx   ecx, _table[ecx]
                shl     dx, 8
                movzx   edx, dx
                shl     eax, 10h
                or      edx, ecx
                or      eax, edx
                retn
f               endp
```

5.1.8 Задача 1.8

Скомпилировано в MSVC 2010 с опцией `/O1`[1]:

```
_a$ = 8         ; size = 4
_b$ = 12        ; size = 4
_c$ = 16        ; size = 4
?s@@YAXPAN00@Z PROC     ; s, COMDAT
    mov     eax, DWORD PTR _b$[esp-4]
    mov     ecx, DWORD PTR _a$[esp-4]
    mov     edx, DWORD PTR _c$[esp-4]
    push    esi
    push    edi
    sub     ecx, eax
    sub     edx, eax
    mov     edi, 200      ; 000000c8H
$LL6@s:
    push    100           ; 00000064H
    pop     esi
$LL3@s:
    fld     QWORD PTR [ecx+eax]
    fadd    QWORD PTR [eax]
    fstp    QWORD PTR [edx+eax]
    add     eax, 8
    dec     esi
    jne     SHORT $LL3@s
    dec     edi
    jne     SHORT $LL6@s
    pop     edi
    pop     esi
    ret     0
?s@@YAXPAN00@Z ENDP     ; s
```

5.1.9 Задача 1.9

Скомпилировано в MSVC 2010 с опцией `/O1`:

```
tv315 = -8              ; size = 4
tv291 = -4              ; size = 4
_a$ = 8                 ; size = 4
_b$ = 12                ; size = 4
```

[1] /O1: оптимизация по размеру кода

```
_c$ = 16                ; size = 4
?m@@YAXPAN00@Z PROC     ; m, COMDAT
    push    ebp
    mov     ebp, esp
    push    ecx
    push    ecx
    mov     edx, DWORD PTR _a$[ebp]
    push    ebx
    mov     ebx, DWORD PTR _c$[ebp]
    push    esi
    mov     esi, DWORD PTR _b$[ebp]
    sub     edx, esi
    push    edi
    sub     esi, ebx
    mov     DWORD PTR tv315[ebp], 100  ; 00000064H
$LL9@m:
    mov     eax, ebx
    mov     DWORD PTR tv291[ebp], 300  ; 0000012cH
$LL6@m:
    fldz
    lea     ecx, DWORD PTR [esi+eax]
    fstp    QWORD PTR [eax]
    mov     edi, 200                  ; 000000c8H
$LL3@m:
    dec     edi
    fld     QWORD PTR [ecx+edx]
    fmul    QWORD PTR [ecx]
    fadd    QWORD PTR [eax]
    fstp    QWORD PTR [eax]
    jne     HORT $LL3@m
    add     eax, 8
    dec     DWORD PTR tv291[ebp]
    jne     SHORT $LL6@m
    add     ebx, 800                 ; 00000320H
    dec     DWORD PTR tv315[ebp]
    jne     SHORT $LL9@m
    pop     edi
    pop     esi
    pop     ebx
    leave
    ret     0
?m@@YAXPAN00@Z ENDP                  ; m
```

5.1.10 Задача 1.10

Если это скомпилировать и запустить, появится некоторое число. Откуда оно берется? Откуда оно берется если скомпилировать в MSVC с оптимизациями (`/Ox`)?

```
#include <stdio.h>

int main()
{
        printf ("%d\n");

        return 0;
};
```

5.2 Средний уровень

5.2.1 Задача 2.1

Довольно известный алгоритм, также включен в стандартную библиотеку Си. Исходник взят из glibc 2.11.1. Скомпилирован в GCC 4.4.1 с ключом `-Os` (оптимизация по размеру кода). Листинг сделан

дизассемблером IDA 4.9 из ELF-файла созданным GCC и линкером.

Для тех кто хочет использовать IDA в процессе изучения, вот здесь лежат .elf и .idb файлы, .idb можно открыть при помощи бесплатой IDA 4.9:

http://conus.info/RE-tasks/middle/1/

```
f               proc near

var_150         = dword ptr -150h
var_14C         = dword ptr -14Ch
var_13C         = dword ptr -13Ch
var_138         = dword ptr -138h
var_134         = dword ptr -134h
var_130         = dword ptr -130h
var_128         = dword ptr -128h
var_124         = dword ptr -124h
var_120         = dword ptr -120h
var_11C         = dword ptr -11Ch
var_118         = dword ptr -118h
var_114         = dword ptr -114h
var_110         = dword ptr -110h
var_C           = dword ptr -0Ch
arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch
arg_8           = dword ptr  10h
arg_C           = dword ptr  14h
arg_10          = dword ptr  18h

                push    ebp
                mov     ebp, esp
                push    edi
                push    esi
                push    ebx
                sub     esp, 14Ch
                mov     ebx, [ebp+arg_8]
                cmp     [ebp+arg_4], 0
                jz      loc_804877D
                cmp     [ebp+arg_4], 4
                lea     eax, ds:0[ebx*4]
                mov     [ebp+var_130], eax
                jbe     loc_804864C
                mov     eax, [ebp+arg_4]
                mov     ecx, ebx
                mov     esi, [ebp+arg_0]
                lea     edx, [ebp+var_110]
                neg     ecx
                mov     [ebp+var_118], 0
                mov     [ebp+var_114], 0
                dec     eax
                imul    eax, ebx
                add     eax, [ebp+arg_0]
                mov     [ebp+var_11C], edx
                mov     [ebp+var_134], ecx
                mov     [ebp+var_124], eax
                lea     eax, [ebp+var_118]
                mov     [ebp+var_14C], eax
                mov     [ebp+var_120], ebx

loc_8048433:                            ; CODE XREF: f+28C
                mov     eax, [ebp+var_124]
                xor     edx, edx
                push    edi
                push    [ebp+arg_10]
                sub     eax, esi
                div     [ebp+var_120]
                push    esi
                shr     eax, 1
```

```
                imul    eax, [ebp+var_120]
                lea     edx, [esi+eax]
                push    edx
                mov     [ebp+var_138], edx
                call    [ebp+arg_C]
                add     esp, 10h
                mov     edx, [ebp+var_138]
                test    eax, eax
                jns     short loc_8048482
                xor     eax, eax

loc_804846D:                            ; CODE XREF: f+CC
                mov     cl, [edx+eax]
                mov     bl, [esi+eax]
                mov     [edx+eax], bl
                mov     [esi+eax], cl
                inc     eax
                cmp     [ebp+var_120], eax
                jnz     short loc_804846D

loc_8048482:                            ; CODE XREF: f+B5
                push    ebx
                push    [ebp+arg_10]
                mov     [ebp+var_138], edx
                push    edx
                push    [ebp+var_124]
                call    [ebp+arg_C]
                mov     edx, [ebp+var_138]
                add     esp, 10h
                test    eax, eax
                jns     short loc_80484F6
                mov     ecx, [ebp+var_124]
                xor     eax, eax

loc_80484AB:                            ; CODE XREF: f+10D
                movzx   edi, byte ptr [edx+eax]
                mov     bl, [ecx+eax]
                mov     [edx+eax], bl
                mov     ebx, edi
                mov     [ecx+eax], bl
                inc     eax
                cmp     [ebp+var_120], eax
                jnz     short loc_80484AB
                push    ecx
                push    [ebp+arg_10]
                mov     [ebp+var_138], edx
                push    esi
                push    edx
                call    [ebp+arg_C]
                add     esp, 10h
                mov     edx, [ebp+var_138]
                test    eax, eax
                jns     short loc_80484F6
                xor     eax, eax

loc_80484E1:                            ; CODE XREF: f+140
                mov     cl, [edx+eax]
                mov     bl, [esi+eax]
                mov     [edx+eax], bl
                mov     [esi+eax], cl
                inc     eax
                cmp     [ebp+var_120], eax
                jnz     short loc_80484E1

loc_80484F6:                            ; CODE XREF: f+ED
                                        ; f+129
```

```
                mov     eax, [ebp+var_120]
                mov     edi, [ebp+var_124]
                add     edi, [ebp+var_134]
                lea     ebx, [esi+eax]
                jmp     short loc_8048513
; ---------------------------------------------------------------------------

loc_804850D:                            ; CODE XREF: f+17B
                add     ebx, [ebp+var_120]

loc_8048513:                            ; CODE XREF: f+157
                                        ; f+1F9
                push    eax
                push    [ebp+arg_10]
                mov     [ebp+var_138], edx
                push    edx
                push    ebx
                call    [ebp+arg_C]
                add     esp, 10h
                mov     edx, [ebp+var_138]
                test    eax, eax
                jns     short loc_8048537
                jmp     short loc_804850D
; ---------------------------------------------------------------------------

loc_8048531:                            ; CODE XREF: f+19D
                add     edi, [ebp+var_134]

loc_8048537:                            ; CODE XREF: f+179
                push    ecx
                push    [ebp+arg_10]
                mov     [ebp+var_138], edx
                push    edi
                push    edx
                call    [ebp+arg_C]
                add     esp, 10h
                mov     edx, [ebp+var_138]
                test    eax, eax
                js      short loc_8048531
                cmp     ebx, edi
                jnb     short loc_8048596
                xor     eax, eax
                mov     [ebp+var_128], edx

loc_804855F:                            ; CODE XREF: f+1BE
                mov     cl, [ebx+eax]
                mov     dl, [edi+eax]
                mov     [ebx+eax], dl
                mov     [edi+eax], cl
                inc     eax
                cmp     [ebp+var_120], eax
                jnz     short loc_804855F
                mov     edx, [ebp+var_128]
                cmp     edx, ebx
                jnz     short loc_8048582
                mov     edx, edi
                jmp     short loc_8048588
; ---------------------------------------------------------------------------

loc_8048582:                            ; CODE XREF: f+1C8
                cmp     edx, edi
                jnz     short loc_8048588
                mov     edx, ebx

loc_8048588:                            ; CODE XREF: f+1CC
                                        ; f+1D0
```

```
                add     ebx, [ebp+var_120]
                add     edi, [ebp+var_134]
                jmp     short loc_80485AB
; ---------------------------------------------------------------------------

loc_8048596:                            ; CODE XREF: f+1A1
                jnz     short loc_80485AB
                mov     ecx, [ebp+var_134]
                mov     eax, [ebp+var_120]
                lea     edi, [ebx+ecx]
                add     ebx, eax
                jmp     short loc_80485B3
; ---------------------------------------------------------------------------

loc_80485AB:                            ; CODE XREF: f+1E0
                                        ; f:loc_8048596
                cmp     ebx, edi
                jbe     loc_8048513

loc_80485B3:                            ; CODE XREF: f+1F5
                mov     eax, edi
                sub     eax, esi
                cmp     eax, [ebp+var_130]
                ja      short loc_80485EB
                mov     eax, [ebp+var_124]
                mov     esi, ebx
                sub     eax, ebx
                cmp     eax, [ebp+var_130]
                ja      short loc_8048634
                sub     [ebp+var_11C], 8
                mov     edx, [ebp+var_11C]
                mov     ecx, [edx+4]
                mov     esi, [edx]
                mov     [ebp+var_124], ecx
                jmp     short loc_8048634
; ---------------------------------------------------------------------------

loc_80485EB:                            ; CODE XREF: f+209
                mov     edx, [ebp+var_124]
                sub     edx, ebx
                cmp     edx, [ebp+var_130]
                jbe     short loc_804862E
                cmp     eax, edx
                mov     edx, [ebp+var_11C]
                lea     eax, [edx+8]
                jle     short loc_8048617
                mov     [edx], esi
                mov     esi, ebx
                mov     [edx+4], edi
                mov     [ebp+var_11C], eax
                jmp     short loc_8048634
; ---------------------------------------------------------------------------

loc_8048617:                            ; CODE XREF: f+252
                mov     ecx, [ebp+var_11C]
                mov     [ebp+var_11C], eax
                mov     [ecx], ebx
                mov     ebx, [ebp+var_124]
                mov     [ecx+4], ebx

loc_804862E:                            ; CODE XREF: f+245
                mov     [ebp+var_124], edi

loc_8048634:                            ; CODE XREF: f+21B
                                        ; f+235 ...
                mov     eax, [ebp+var_14C]
```

```
                cmp     [ebp+var_11C], eax
                ja      loc_8048433
                mov     ebx, [ebp+var_120]

loc_804864C:                            ; CODE XREF: f+2A
                mov     eax, [ebp+arg_4]
                mov     ecx, [ebp+arg_0]
                add     ecx, [ebp+var_130]
                dec     eax
                imul    eax, ebx
                add     eax, [ebp+arg_0]
                cmp     ecx, eax
                mov     [ebp+var_120], eax
                jbe     short loc_804866B
                mov     ecx, eax

loc_804866B:                            ; CODE XREF: f+2B3
                mov     esi, [ebp+arg_0]
                mov     edi, [ebp+arg_0]
                add     esi, ebx
                mov     edx, esi
                jmp     short loc_80486A3
; ---------------------------------------------------------------------------

loc_8048677:                            ; CODE XREF: f+2F1
                push    eax
                push    [ebp+arg_10]
                mov     [ebp+var_138], edx
                mov     [ebp+var_13C], ecx
                push    edi
                push    edx
                call    [ebp+arg_C]
                add     esp, 10h
                mov     edx, [ebp+var_138]
                mov     ecx, [ebp+var_13C]
                test    eax, eax
                jns     short loc_80486A1
                mov     edi, edx

loc_80486A1:                            ; CODE XREF: f+2E9
                add     edx, ebx

loc_80486A3:                            ; CODE XREF: f+2C1
                cmp     edx, ecx
                jbe     short loc_8048677
                cmp     edi, [ebp+arg_0]
                jz      loc_8048762
                xor     eax, eax

loc_80486B2:                            ; CODE XREF: f+313
                mov     ecx, [ebp+arg_0]
                mov     dl, [edi+eax]
                mov     cl, [ecx+eax]
                mov     [edi+eax], cl
                mov     ecx, [ebp+arg_0]
                mov     [ecx+eax], dl
                inc     eax
                cmp     ebx, eax
                jnz     short loc_80486B2
                jmp     loc_8048762
; ---------------------------------------------------------------------------

loc_80486CE:                            ; CODE XREF: f+3C3
                lea     edx, [esi+edi]
                jmp     short loc_80486D5
; ---------------------------------------------------------------------------
```

```
loc_80486D3:                            ; CODE XREF: f+33B
                add     edx, edi

loc_80486D5:                            ; CODE XREF: f+31D
                push    eax
                push    [ebp+arg_10]
                mov     [ebp+var_138], edx
                push    edx
                push    esi
                call    [ebp+arg_C]
                add     esp, 10h
                mov     edx, [ebp+var_138]
                test    eax, eax
                js      short loc_80486D3
                add     edx, ebx
                cmp     edx, esi
                mov     [ebp+var_124], edx
                jz      short loc_804876F
                mov     edx, [ebp+var_134]
                lea     eax, [esi+ebx]
                add     edx, eax
                mov     [ebp+var_11C], edx
                jmp     short loc_804875B
; ---------------------------------------------------------------------------

loc_8048710:                            ; CODE XREF: f+3AA
                mov     cl, [eax]
                mov     edx, [ebp+var_11C]
                mov     [ebp+var_150], eax
                mov     byte ptr [ebp+var_130], cl
                mov     ecx, eax
                jmp     short loc_8048733
; ---------------------------------------------------------------------------

loc_8048728:                            ; CODE XREF: f+391
                mov     al, [edx+ebx]
                mov     [ecx], al
                mov     ecx, [ebp+var_128]

loc_8048733:                            ; CODE XREF: f+372
                mov     [ebp+var_128], edx
                add     edx, edi
                mov     eax, edx
                sub     eax, edi
                cmp     [ebp+var_124], eax
                jbe     short loc_8048728
                mov     dl, byte ptr [ebp+var_130]
                mov     eax, [ebp+var_150]
                mov     [ecx], dl
                dec     [ebp+var_11C]

loc_804875B:                            ; CODE XREF: f+35A
                dec     eax
                cmp     eax, esi
                jnb     short loc_8048710
                jmp     short loc_804876F
; ---------------------------------------------------------------------------

loc_8048762:                            ; CODE XREF: f+2F6
                                        ; f+315
                mov     edi, ebx
                neg     edi
                lea     ecx, [edi-1]
                mov     [ebp+var_134], ecx
```

```
loc_804876F:                                ; CODE XREF: f+347
                                            ; f+3AC
                add     esi, ebx
                cmp     esi, [ebp+var_120]
                jbe     loc_80486CE

loc_804877D:                                ; CODE XREF: f+13
                lea     esp, [ebp-0Ch]
                pop     ebx
                pop     esi
                pop     edi
                pop     ebp
                retn
f               endp
```

5.3 crackme / keygenme

/IFRUНесколько моих keygenme[2]: Couple of my keygenmes[3]:

http://crackmes.de/users/yonkie/

[2] программа имитирующая защиту вымышленной программы, для которой нужно сделать генератор ключей/лицензий.

[3] program which imitates fictional software protection, for which one need to make a keys/licenses generator

Глава 6

Инструменты

- IDA как дизассемблер. Старая бесплатная версия доступна для скачивания: http://www.hex-rays.com/idapro/idadownfreeware.htm.
- Microsoft Visual Studio Express[1]: Усеченная версия Visual Studio, пригодная для простых экспериментов.
- Hiew[2] /IFRUдля мелкой модификации кода в исполняемых файлах for small modifications of code in binary files.

6.0.1 Отладчик

tracer[3] вместо отладчика.

Со временем я отказался использовать отладчик, потому что все что мне нужно от него: это иногда подсмотреть какие-либо аргументы какой-либо функции во время исполнения или состояние регистров в определенном месте. Каждый раз загружать отладчик для этого это слишком, поэтому я написал очень простую утилиту *tracer*. Она консольная, запускается из командной строки, позволяет перехватывать исполнение функций, ставить брякпоинты на произвольные места, смотреть состояние регистров, модифицировать их, и так далее.

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, итд.

[1] http://www.microsoft.com/express/Downloads/
[2] http://www.hiew.ru/
[3] http://conus.info/gt/

Глава 7

Что стоит почитать

7.1 Книги

7.1.1 Windows

- Windows® Internals (Mark E. Russinovich and David A. Solomon with Alex Ionescu)[1]

7.1.2 Си/Си++

- Стандарт языка Си++: ISO/IEC 14882:2003[2]

7.1.3 x86 / x86-64

- Документация от Intel: http://www.intel.com/products/processor/manuals/
- Документация от AMD: http://developer.amd.com/documentation/guides/Pages/default.aspx#manuals

7.2 Блоги

7.2.1 Windows

- Microsoft: Raymond Chen
- http://www.nynaeve.net/

[1] http://www.microsoft.com/learning/en/us/book.aspx?ID=12069&locale=en-us
[2] http://www.iso.org/iso/catalogue_detail.htm?csnumber=38110

Глава 8

Прочее

8.1 Еще примеры

- (eng) http://conus.info/RE-articles/qr9.html
- (eng) http://conus.info/RE-articles/sapgui.html

8.2 Аномалии компиляторов

Intel C++ 10.1 которым скомпилирован Oracle RDBMS 11.2 Linux86, может сгенерировать два JZ идущих подряд, причем на второй JZ нет ссылки ниоткуда. Второй JZ таким образом, не имеет никакого смысла.

Например, kdli.o из libserver11.a:

```
.text:08114CF1                    loc_8114CF1:                              ; CODE XREF:
    __PGOSF539_kdlimemSer+89A
.text:08114CF1                                                              ;
    __PGOSF539_kdlimemSer+3994
.text:08114CF1 8B 45 08                          mov     eax, [ebp+arg_0]
.text:08114CF4 0F B6 50 14                       movzx   edx, byte ptr [eax+14h]
.text:08114CF8 F6 C2 01                          test    dl, 1
.text:08114CFB 0F 85 17 08 00 00                 jnz     loc_8115518
.text:08114D01 85 C9                             test    ecx, ecx
.text:08114D03 0F 84 8A 00 00 00                 jz      loc_8114D93
.text:08114D09 0F 84 09 08 00 00                 jz      loc_8115518
.text:08114D0F 8B 53 08                          mov     edx, [ebx+8]
.text:08114D12 89 55 FC                          mov     [ebp+var_4], edx
.text:08114D15 31 C0                             xor     eax, eax
.text:08114D17 89 45 F4                          mov     [ebp+var_C], eax
.text:08114D1A 50                                push    eax
.text:08114D1B 52                                push    edx
.text:08114D1C E8 03 54 00 00                    call    len2nbytes
.text:08114D21 83 C4 08                          add     esp, 8
```

Еще там же:

```
.text:0811A2A5                    loc_811A2A5:                              ; CODE XREF:
    kdliSerLengths+11C
.text:0811A2A5                                                              ; kdliSerLengths
    +1C1
.text:0811A2A5 8B 7D 08                          mov     edi, [ebp+arg_0]
.text:0811A2A8 8B 7F 10                          mov     edi, [edi+10h]
.text:0811A2AB 0F B6 57 14                       movzx   edx, byte ptr [edi+14h]
.text:0811A2AF F6 C2 01                          test    dl, 1
.text:0811A2B2 75 3E                             jnz     short loc_811A2F2
.text:0811A2B4 83 E0 01                          and     eax, 1
.text:0811A2B7 74 1F                             jz      short loc_811A2D8
.text:0811A2B9 74 37                             jz      short loc_811A2F2
.text:0811A2BB 6A 00                             push    0
```

```
.text:0811A2BD FF 71 08                                push    dword ptr [ecx+8]
.text:0811A2C0 E8 5F FE FF FF                          call    len2nbytes
```

Возможно, это ошибка его кодегенератора, не выявленная тестами (ведь результирующий код и так работает нормально).

Глава 9

Ответы на задачи

9.1 Легкий уровень

9.1.1 Задача 1.1

Решение: `toupper()`.

Исходник на Си:

```
char toupper ( char c )
{
    if( c >= 'a' && c <= 'z' ) {
        c = c - 'a' + 'A';
    }
    return( c );
}
```

9.1.2 Задача 1.2

Ответ: `atoi()`

Исходник на Си:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int atoi ( const *p )  /* convert ASCII string to integer */
{
    int i;
    char s;

    while( isspace ( *p ) )
        ++p;
    s = *p;
    if( s == '+' || s == '-' )
        ++p;
    i = 0;
    while( isdigit(*p) ) {
        i = i * 10 + *p - '0';
        ++p;
    }
    if( s == '-' )
        i = - i;
    return( i );
}
```

9.1.3 Задача 1.3

Ответ: `srand()` / `rand()`.

Исходник на Си:

```
static unsigned int v;

void srand (unsigned int s)
{
        v = s;
}

int rand ()
{
        return( ((v = v * 214013L
            + 2531011L) >> 16) & 0x7fff );
}
```

9.1.4 Задача 1.4

Ответ: `strstr()`.

Исходник на Си:

```
char * strstr (
        const char * str1,
        const char * str2
        )
{
        char *cp = (char *) str1;
        char *s1, *s2;

        if ( !*str2 )
            return((char *)str1);

        while (*cp)
        {
                s1 = cp;
                s2 = (char *) str2;

                while ( *s1 && *s2 && !(*s1-*s2) )
                        s1++, s2++;

                if (!*s2)
                        return(cp);

                cp++;
        }

        return(NULL);

}
```

9.1.5 Задача 1.5

Подсказка #1: Не забывайте что `__v` — глобальная переменная.

Подсказка #2: Эта функция вызывается startup-кодом перед вызовом `main()`.

Ответ: это проверка на наличие FDIV-ошибки в ранних процессорах Pentium[1].

Исходник на Си:

```
unsigned _v; // _v

enum e {
    PROB_P5_DIV = 0x0001
};
```

[1] http://en.wikipedia.org/wiki/Pentium_FDIV_bug

```
void f( void ) // __verify_pentium_fdiv_bug
{
    /*
        Verify we have got the Pentium FDIV problem.
        The volatiles are to scare the optimizer away.
    */
    volatile double     v1      = 4195835;
    volatile double     v2    = 3145727;

    if( (v1 - (v1/v2)*v2) > 1.0e-8 ) {
        _v |= PROB_P5_DIV;
    }
}
```

9.1.6 Задача 1.6

Подсказка: если погуглить применяемую здесь константу, это может помочь.

Ответ: шифрование алгоритмом TEA[2].

Исходник на Си (взято с http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm):

```
void f (unsigned int* v, unsigned int* k) {
    unsigned int v0=v[0], v1=v[1], sum=0, i;           /* set up */
    unsigned int delta=0x9e3779b9;                     /* a key schedule constant */
    unsigned int k0=k[0], k1=k[1], k2=k[2], k3=k[3];   /* cache key */
    for (i=0; i < 32; i++) {                       /* basic cycle start */
        sum += delta;
        v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
    }                                              /* end cycle */
    v[0]=v0; v[1]=v1;
}
```

9.1.7 Задача 1.7

Подсказка: таблица содержит заранее вычисленные значения. Можно было бы обойтись и без нее, но тогда функция работала бы чуть медленнее.

Ответ: эта функция переставляет все биты во входном 32-битном слове наоборот. Это `lib/bitrev.c` из ядра Linux.

Исходник на Си:

```
const unsigned char byte_rev_table[256] = {
        0x00, 0x80, 0x40, 0xc0, 0x20, 0xa0, 0x60, 0xe0,
        0x10, 0x90, 0x50, 0xd0, 0x30, 0xb0, 0x70, 0xf0,
        0x08, 0x88, 0x48, 0xc8, 0x28, 0xa8, 0x68, 0xe8,
        0x18, 0x98, 0x58, 0xd8, 0x38, 0xb8, 0x78, 0xf8,
        0x04, 0x84, 0x44, 0xc4, 0x24, 0xa4, 0x64, 0xe4,
        0x14, 0x94, 0x54, 0xd4, 0x34, 0xb4, 0x74, 0xf4,
        0x0c, 0x8c, 0x4c, 0xcc, 0x2c, 0xac, 0x6c, 0xec,
        0x1c, 0x9c, 0x5c, 0xdc, 0x3c, 0xbc, 0x7c, 0xfc,
        0x02, 0x82, 0x42, 0xc2, 0x22, 0xa2, 0x62, 0xe2,
        0x12, 0x92, 0x52, 0xd2, 0x32, 0xb2, 0x72, 0xf2,
        0x0a, 0x8a, 0x4a, 0xca, 0x2a, 0xaa, 0x6a, 0xea,
        0x1a, 0x9a, 0x5a, 0xda, 0x3a, 0xba, 0x7a, 0xfa,
        0x06, 0x86, 0x46, 0xc6, 0x26, 0xa6, 0x66, 0xe6,
        0x16, 0x96, 0x56, 0xd6, 0x36, 0xb6, 0x76, 0xf6,
        0x0e, 0x8e, 0x4e, 0xce, 0x2e, 0xae, 0x6e, 0xee,
        0x1e, 0x9e, 0x5e, 0xde, 0x3e, 0xbe, 0x7e, 0xfe,
        0x01, 0x81, 0x41, 0xc1, 0x21, 0xa1, 0x61, 0xe1,
        0x11, 0x91, 0x51, 0xd1, 0x31, 0xb1, 0x71, 0xf1,
        0x09, 0x89, 0x49, 0xc9, 0x29, 0xa9, 0x69, 0xe9,
        0x19, 0x99, 0x59, 0xd9, 0x39, 0xb9, 0x79, 0xf9,
```

[2] Tiny Encryption Algorithm

```
        0x05, 0x85, 0x45, 0xc5, 0x25, 0xa5, 0x65, 0xe5,
        0x15, 0x95, 0x55, 0xd5, 0x35, 0xb5, 0x75, 0xf5,
        0x0d, 0x8d, 0x4d, 0xcd, 0x2d, 0xad, 0x6d, 0xed,
        0x1d, 0x9d, 0x5d, 0xdd, 0x3d, 0xbd, 0x7d, 0xfd,
        0x03, 0x83, 0x43, 0xc3, 0x23, 0xa3, 0x63, 0xe3,
        0x13, 0x93, 0x53, 0xd3, 0x33, 0xb3, 0x73, 0xf3,
        0x0b, 0x8b, 0x4b, 0xcb, 0x2b, 0xab, 0x6b, 0xeb,
        0x1b, 0x9b, 0x5b, 0xdb, 0x3b, 0xbb, 0x7b, 0xfb,
        0x07, 0x87, 0x47, 0xc7, 0x27, 0xa7, 0x67, 0xe7,
        0x17, 0x97, 0x57, 0xd7, 0x37, 0xb7, 0x77, 0xf7,
        0x0f, 0x8f, 0x4f, 0xcf, 0x2f, 0xaf, 0x6f, 0xef,
        0x1f, 0x9f, 0x5f, 0xdf, 0x3f, 0xbf, 0x7f, 0xff,
};

unsigned char bitrev8(unsigned char byte)
{
        return byte_rev_table[byte];
}

unsigned short bitrev16(unsigned short x)
{
        return (bitrev8(x & 0xff) << 8) | bitrev8(x >> 8);
}

/**
 * bitrev32 - reverse the order of bits in a unsigned int value
 * @x: value to be bit-reversed
 */

unsigned int bitrev32(unsigned int x)
{
        return (bitrev16(x & 0xffff) << 16) | bitrev16(x >> 16);
}
```

9.1.8 Задача 1.8

Ответ: сложение двух матриц размером 100 на 200 элементов типа *double*.

Исходник на Си/Си++:

```
#define M    100
#define N    200

void s(double *a, double *b, double *c)
{
  for(int i=0;i<N;i++)
    for(int j=0;j<M;j++)
      *(c+i*M+j)=*(a+i*M+j) + *(b+i*M+j);
};
```

9.1.9 Задача 1.9

Ответ: умножение двух матриц размерами 100*200 и 100*300 элементов типа *double*, результат: матрица 100*300.

Исходник на Си/Си++:

```
#define M     100
#define N     200
#define P     300

void m(double *a, double *b, double *c)
{
  for(int i=0;i<M;i++)
    for(int j=0;j<P;j++)
    {
```

```
      *(c+i*M+j)=0;
      for (int k=0;k<N;k++) *(c+i*M+j)+=*(a+i*M+j) * *(b+i*M+j);
    }
};
```

9.2 Средний уровень

9.2.1 Задача 2.1

Подсказка #1: В этом коде есть одна особенность, по которой можно значительно сузить поиск функции в glibc..

Ответ: особенность — это вызов callback-функции 2.17, указатель на которую передается в четвертом аргументе. Это `quicksort()`.

Исходник на Си:

```
/* Copyright (C) 1991,1992,1996,1997,1999,2004 Free Software Foundation, Inc.
   This file is part of the GNU C Library.
   Written by Douglas C. Schmidt (schmidt@ics.uci.edu).

   The GNU C Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   The GNU C Library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.

   You should have received a copy of the GNU Lesser General Public
   License along with the GNU C Library; if not, write to the Free
   Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
   02111-1307 USA.  */

/* If you consider tuning this algorithm, you should consult first:
   Engineering a sort function; Jon Bentley and M. Douglas McIlroy;
   Software - Practice and Experience; Vol. 23 (11), 1249-1265, 1993.  */

#include <alloca.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>

typedef int (*__compar_d_fn_t) (__const void *, __const void *, void *);

/* Byte-wise swap two items of size SIZE. */
#define SWAP(a, b, size)                        \
do                                              \
{                                               \
    register size_t __size = (size);            \
    register char *__a = (a), *__b = (b);       \
    do                                          \
    {                                           \
        char __tmp = *__a;                      \
        *__a++ = *__b;                          \
        *__b++ = __tmp;                         \
    } while (--__size > 0);                     \
} while (0)

/* Discontinue quicksort algorithm when partition gets below this size.
   This particular magic number was chosen to work best on a Sun 4/260. */
#define MAX_THRESH 4

/* Stack node declarations used to store unfulfilled partition obligations. */
```

```
typedef struct
{
    char *lo;
    char *hi;
} stack_node;

/* The next 4 #defines implement a very fast in-line stack abstraction. */
/* The stack needs log (total_elements) entries (we could even subtract
   log(MAX_THRESH)).  Since total_elements has type size_t, we get as
   upper bound for log (total_elements):
   bits per byte (CHAR_BIT) * sizeof(size_t).  */
#define STACK_SIZE    (CHAR_BIT * sizeof(size_t))
#define PUSH(low, high)    ((void) ((top->lo = (low)), (top->hi = (high)), ++top))
#define    POP(low, high)    ((void) (--top, (low = top->lo), (high = top->hi)))
#define    STACK_NOT_EMPTY    (stack < top)


/* Order size using quicksort.  This implementation incorporates
   four optimizations discussed in Sedgewick:

   1. Non-recursive, using an explicit stack of pointer that store the
      next array partition to sort.  To save time, this maximum amount
      of space required to store an array of SIZE_MAX is allocated on the
      stack.  Assuming a 32-bit (64 bit) integer for size_t, this needs
      only 32 * sizeof(stack_node) == 256 bytes (for 64 bit: 1024 bytes).
      Pretty cheap, actually.

   2. Chose the pivot element using a median-of-three decision tree.
      This reduces the probability of selecting a bad pivot value and
      eliminates certain extraneous comparisons.

   3. Only quicksorts TOTAL_ELEMS / MAX_THRESH partitions, leaving
      insertion sort to order the MAX_THRESH items within each partition.
      This is a big win, since insertion sort is faster for small, mostly
      sorted array segments.

   4. The larger of the two sub-partitions is always pushed onto the
      stack first, with the algorithm then concentrating on the
      smaller partition.  This *guarantees* no more than log (total_elems)
      stack size is needed (actually O(1) in this case)!  */

void
_quicksort (void *const pbase, size_t total_elems, size_t size,
        __compar_d_fn_t cmp, void *arg)
{
    register char *base_ptr = (char *) pbase;

    const size_t max_thresh = MAX_THRESH * size;

    if (total_elems == 0)
        /* Avoid lossage with unsigned arithmetic below.  */
        return;

    if (total_elems > MAX_THRESH)
    {
        char *lo = base_ptr;
        char *hi = &lo[size * (total_elems - 1)];
        stack_node stack[STACK_SIZE];
        stack_node *top = stack;

        PUSH (NULL, NULL);

        while (STACK_NOT_EMPTY)
        {
            char *left_ptr;
            char *right_ptr;
```

```
            /* Select median value from among LO, MID, and HI. Rearrange
               LO and HI so the three values are sorted. This lowers the
               probability of picking a pathological pivot value and
               skips a comparison for both the LEFT_PTR and RIGHT_PTR in
               the while loops. */

            char *mid = lo + size * ((hi - lo) / size >> 1);

            if ((*cmp) ((void *) mid, (void *) lo, arg) < 0)
                SWAP (mid, lo, size);
            if ((*cmp) ((void *) hi, (void *) mid, arg) < 0)
                SWAP (mid, hi, size);
            else
                goto jump_over;
            if ((*cmp) ((void *) mid, (void *) lo, arg) < 0)
                SWAP (mid, lo, size);
jump_over:;

          left_ptr  = lo + size;
          right_ptr = hi - size;

          /* Here's the famous ``collapse the walls'' section of quicksort.
             Gotta like those tight inner loops!  They are the main reason
             that this algorithm runs much faster than others. */
          do
          {
              while ((*cmp) ((void *) left_ptr, (void *) mid, arg) < 0)
                  left_ptr += size;

              while ((*cmp) ((void *) mid, (void *) right_ptr, arg) < 0)
                  right_ptr -= size;

              if (left_ptr < right_ptr)
              {
                  SWAP (left_ptr, right_ptr, size);
                  if (mid == left_ptr)
                      mid = right_ptr;
                  else if (mid == right_ptr)
                      mid = left_ptr;
                  left_ptr += size;
                  right_ptr -= size;
              }
              else if (left_ptr == right_ptr)
              {
                  left_ptr += size;
                  right_ptr -= size;
                  break;
              }
          }
          while (left_ptr <= right_ptr);

          /* Set up pointers for next iteration.  First determine whether
             left and right partitions are below the threshold size.  If so,
             ignore one or both.  Otherwise, push the larger partition's
             bounds on the stack and continue sorting the smaller one. */

          if ((size_t) (right_ptr - lo) <= max_thresh)
          {
              if ((size_t) (hi - left_ptr) <= max_thresh)
                  /* Ignore both small partitions. */
                  POP (lo, hi);
              else
                  /* Ignore small left partition. */
                  lo = left_ptr;
          }
```

```
          else if ((size_t) (hi - left_ptr) <= max_thresh)
              /* Ignore small right partition. */
              hi = right_ptr;
          else if ((right_ptr - lo) > (hi - left_ptr))
          {
              /* Push larger left partition indices. */
              PUSH (lo, right_ptr);
              lo = left_ptr;
          }
          else
          {
              /* Push larger right partition indices. */
              PUSH (left_ptr, hi);
              hi = right_ptr;
          }
        }
    }

    /* Once the BASE_PTR array is partially sorted by quicksort the rest
       is completely sorted using insertion sort, since this is efficient
       for partitions below MAX_THRESH size. BASE_PTR points to the beginning
       of the array to sort, and END_PTR points at the very last element in
       the array (*not* one beyond it!). */

#define min(x, y) ((x) < (y) ? (x) : (y))

    {
        char *const end_ptr = &base_ptr[size * (total_elems - 1)];
        char *tmp_ptr = base_ptr;
        char *thresh = min(end_ptr, base_ptr + max_thresh);
        register char *run_ptr;

        /* Find smallest element in first threshold and place it at the
           array's beginning.  This is the smallest array element,
           and the operation speeds up insertion sort's inner loop. */

        for (run_ptr = tmp_ptr + size; run_ptr <= thresh; run_ptr += size)
            if ((*cmp) ((void *) run_ptr, (void *) tmp_ptr, arg) < 0)
                tmp_ptr = run_ptr;

        if (tmp_ptr != base_ptr)
            SWAP (tmp_ptr, base_ptr, size);

        /* Insertion sort, running from left-hand-side up to right-hand-side.  */

        run_ptr = base_ptr + size;
        while ((run_ptr += size) <= end_ptr)
        {
            tmp_ptr = run_ptr - size;
            while ((*cmp) ((void *) run_ptr, (void *) tmp_ptr, arg) < 0)
                tmp_ptr -= size;

            tmp_ptr += size;
            if (tmp_ptr != run_ptr)
            {
                char *trav;

                trav = run_ptr + size;
                while (--trav >= run_ptr)
                {
                    char c = *trav;
                    char *hi, *lo;

                    for (hi = lo = trav; (lo -= size) >= tmp_ptr; hi = lo)
                        *hi = *lo;
                    *hi = c;
```

```
                    }
                }
            }
        }
}
```