Quick introduction to reverse engineering for beginners

Dennis Yurichev
<dennis@yurichev.com>

Text version (August 15, 2013).
Probably, newer version of this text, and also Russian language version is also accessible at
http://yurichev.com/RE-book.html
You may also subscribe to my twitter, to get information about updates of this text, etc: @yurichev

Contents

Preface

Here (will be) some of my notes about reverse engineering in English language for those beginners who like to learn to understand x86 (which is a most large mass of all executable software in the world) and ARM code created by C/C++ compilers.

There are two most used assembly language syntax: Intel (most used in DOS/Windows) and AT&T (used in *NIX) [1]. Here we use Intel syntax. IDA 5 produce Intel syntax listings too.

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

About author

Dennis Yurichev is experienced reverse engineering, available for hire as reverse engineer, consultant, trainer. His CV is available here.

Thanks

Andrey ”herm1t” Baranovich, Slava ”Avid” Kazakov, Stanislav ”Beaver” Bobrytskyy, Alexander Lysenko, Alexander ”Lstar” Chernenkiy, Arnaud Patard (rtp on #debian-arm IRC).

Chapter 1

Compiler's patterns

When I first learn C and then C++, I was just writing small pieces of code, compiling it and see what is producing in assembly language, that was an easy way to me. I did it a lot times and relation between C/C++ code and what compiler produce was imprinted in my mind so deep so that I can quickly understand what was in C code when I look at produced x86 code. Perhaps, this method may be helpful for anyone else, so I'll try to describe here some examples.

1.1 Hello, world!

Let's start with famous example from the book "The C programming Language" [Ker88]:

```
#include <stdio.h>

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

1.1.1 x86

MSVC

Let's compile it in MSVC 2010: `cl 1.cpp /Fa1.asm`
(/Fa option mean generate assembly listing file)

Listing 1.1: MSVC 2010

```
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
```

Compiler generated `1.obj` file which will be linked into `1.exe`.
In our case, the file contain two segments: `CONST` (for data constants) and `_TEXT` (for code).
The string "`hello, world`" in C/C++ has type `const char*`, however hasn't its own name.
But compiler need to work with the string somehow, so it define internal name `$SG3830` for it.
As we can see, the string is terminated by zero byte — it's C/C++ standard for strings.
In the code segment `_TEXT` there are only one function so far — `main()`.
Function `main()` starting with prologue code and ending with epilogue code, like almost any function.
Read more about it in section about function prolog and epilog 2.2.
After function prologue we see a function `printf()` call: `CALL _printf`.
Before the call, string address (or pointer to it) containing our greeting is placed into stack with help of `PUSH` instruction.
When `printf()` function returning control flow to `main()` function, string address (or pointer to it) is still in stack.
Because we do not need it anymore, stack pointer (`ESP` register) is to be corrected.
`ADD ESP, 4` mean add 4 to the value in `ESP` register.
Why 4? Since it is 32-bit code, we need exactly 4 bytes for address passing through the stack. It's 8 bytes in x64-code
"`ADD ESP, 4`" is equivalent to "`POP register`" but without any register usage[1].

[1] CPU flags, however, modified

Some compilers like Intel C++ Compiler, at the same point, could emit POP ECX instead of ADD (for example, such pattern can be observed in Oracle RDBMS code, compiled by Intel C++ compiler), and this instruction has almost the same effect, but ECX register contents will be rewritten.

Probably, Intel C++ compiler using POP ECX because this instruction's opcode is shorter then ADD ESP, x (1 byte against 3).

Read more about stack in relevant section 1.2.

After printf() call, in original C/C++ code was return 0 — return zero as a main() function result.

In the generated code this is implemented by instruction XOR EAX, EAX

XOR, in fact, just "eXclusive OR" [2], but compilers using it often instead of MOV EAX, 0 — slightly shorter opcode again (2 bytes against 5).

Some compilers emitting SUB EAX, EAX, which mean *SUBtract EAX value from EAX*, which is in any case will result zero.

Last instruction RET returning control flow to calling function. Usually, it's C/C++ CRT[3] code, which, in turn, return control to operation system.

GCC

Now let's try to compile the same C/C++ code in GCC 4.4.1 compiler in Linux: gcc 1.c -o 1

After, with the IDA 5 disassembler assistance, let's see how main() function was created.

Note: we could also see GCC assembler result applying option -S -masm=intel

Listing 1.2: GCC

```
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
```

Almost the same. Address of "hello world" string (stored in data segment) is saved in EAX register first, then it stored into stack. Also, in function prologue we see AND ESP, 0FFFFFFF0h — this instruction aligning ESP value on 16-byte border, resulting all values in stack aligned too (CPU performing better if values it working with are located in memory at addresses aligned by 4 or 16 byte border)[4].

SUB ESP, 10h allocate 16 bytes in stack, although, as we could see below, only 4 need here.

This is because the size of allocated stack is also aligned on 16-byte border.

String address (or pointer to string) is then writing directly into stack space without PUSH instruction use. *var_10* — is local variable, but also argument for printf(). Read below about it.

Then printf() function is called.

Unlike MSVC, GCC while compiling without optimization turned on, emitting MOV EAX, 0 instead of shorter opcode.

The last instruction LEAVE — is MOV ESP, EBP and POP EBP instructions pair equivalent — in other words, this instruction setting back stack pointer (ESP) and EBP register to its initial state.

This is necessary because we modified these register values (ESP and EBP) at the function start (executing MOV EBP, ESP / AND ESP, ...).

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

[3] C Run-Time Code

[4] Wikipedia: Data structure alignment

1.1.2 ARM

For my experiments with ARM CPU I choose two compilers: popular in embedded area Keil Release 6/2013 and Apple Xcode 4.6.3 IDE (with LLVM-GCC 4.2 compiler), producing code for ARM-comptaible processors and SoCs[5] in iPod/iPhone/iPad, Windows 8 and Window RT tables[6] and also such devices as Raspberry Pi.

Non-optimizing Keil + ARM mode

Let's start by compiling our example in Keil:

```
armcc.exe --arm --c90 -O0 1.c
```

armcc compiler producing assembly listing, but it has some high-level ARM-processor related macros[7], but it's more important for us to see instructions "as is", so let's see compiled results in IDA 5.

Listing 1.3: Non-optimizing Keil + ARM mode + IDA 5

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

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

Here is couple of ARM-related facts we should know in order to proceed. ARM processor has at least two major modes: ARM mode and thumb. In first (ARM) mode all instructions are enabled and each has 32-bit (4 bytes) size. In second (thumb) mode each instruction has 16-bit (or 2 bytes) size[8]. Thumb mode may look attractive because program in it may be 1) compact; 2) executing faster on microcontroller having 16-bit memory datapath. Nothing come for free of charge, so, in thumb mode there are reduced instruction set, only 8 registers are accessible and one need several thumb instructions for doing some operations when in ARM mode you'll need just one. Starting at ARMv7, there are also thumb-2 instructions set present, this is a thumb extended to support much bigger instructions set. There is a common misconception that thumb-2 is a mix of ARM and thumb. It's not correct. But rather thumb-2 was extended to support processor features so fully, so now it can compete with ARM mode. A program for ARM processor may be mix of procedures compiled for both modes. Majority of iPod/iPhone/iPad applications are compiled for thumb-2 instructions set, because Xcode do this by default.

In example we see here we can easily see that each instruction has size of 4 bytes. Indeed, we compiled our code for ARM mode, but for thumb.

The very first instruction ''`STMFD SP!, {R4,LR}`''[9] works here as `PUSH` in x86, instruction, writing values of two (`R4` and `LR`) registers into stack. Indeed, in output listing, *armcc* compiler, for the sake of simplification, showing here ''`PUSH {r4,lr}`'' instruction. But it's not quite correct, `PUSH` instruction available only in thumb mode, so, to make things less messy, I offered to work in IDA 5.

So this instruction writes values of `R4` and `LR` registers at the address in memory to which `SP`[10] pointing, then decrements `SP` so it will points to a place in stack free for new entries.

This instruction, like `PUSH` instruction in thumb mode, is able save several register values at once and this may be useful. By the way, there is no such thing in x86. It's also can be noted that `STMFD` — generalization of `PUSH` instruction (extending its features), because it can work with any register, not just with `SP` and this can be very useful.

''`ADR R0, aHelloWorld`'' instruction adding `PC` register value to the offset, where the *"hello, world"* string is located. How `PC` register used here, one might ask? This is so called "position-independent code" [11], it is intended to be executed not to be fixed to any addresses in memory. In the opcode of `ADR` instruction, here is encoded a

[5] system on chip
[6] http://en.wikipedia.org/wiki/List_of_Windows_8_and_RT_tablet_devices
[7] for example, ARM mode lacks `PUSH`/`POP` instructions
[8] NOTTRANSLATED
[9] Store Multiple Full Descending
[10] stack pointer
[11] Read more about it in relevant section 2.6

difference between address of this instruction and the place where the string is located. Difference will always be constant, without any dependence to the address where that code being loaded, by operation system, presumably. That's why all we need is to add address of current instruction (from PC) in order to get absolute address of our C-string in memory.

''BL __2printf''[12] instruction calling printf() function. That's how this instruction works:

- write address after BL instruction ($0xC$) into LR[13] register;
- then pass control flow into printf() by writing its address into PC[14] register.

Because, when printf() finishes its work, it should have information, where it should return control, that's why each function passes control to the address stored in LR register.

That is the difference between "pure" RISC-processors like ARM and x86, where address of return is stored in stack[15].

By the way, absolute 32-bit address or offset cannot be encoded in 32-bit BL instruction, because it has space only for 24 bits. It's also worth to note that all ARM mode instructions has size 4 bytes (32 bits), hence they all can be located only on 4-byte boundary addresses. This mean, last 2 bits of instruction address (always zero bits) may be omitted. In summary, we have 26 bit for offset encoding, this is enough to represent offset $\pm \approx 32M$.

Next ''MOV R0, #0''[16] instruction just writes 0 into R0 register. That's because our C-function returning 0 and returning value is to be placed in R0.

The last instruction ''LDMFD SP!, R4,PC''[17] is an inversive instruction of STMFD, it loads values from stack for saving them into R4 and PC, incremeting stack pointer SP. It can be said, it is similar to POP. Note: the very first instruction STMFD saved R4 and LR into stack, but R4 and PC are *restored*. As I wrote before, in LR[18] register address of place saved, to where each function should return control. The very first function saving its value in stack because our main() function will use that register in order to call printf(). And then, in the function end this value can be written to PC, thus, by passing control to where our function was called. Since our main() function is usually primary function in C/C++, apparently, control will be returned to operation system loader or to some place in runtime C functions, or something like that.

DCB — assembly language directive, defining array of bytes or ASCII-strings, similar to DB directive in x86-assembly language.

Non-optimizing Keil: thumb mode

Let's compile the same example in Keil in thumb mode:

```
armcc.exe --thumb --c90 -O0 1.c
```

We will get (in IDA 5):

Listing 1.4: Non-optimizing Keil + thumb mode + IDA 5

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

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

We can easily spot 2-byte (16-bit) opcodes, this is, as I mentioned, thumb. Except BL instruction. In fact, it consisted in two 16-bit instructions. That's because it's not possible to load offset to printf() function into PC when using so small space in one 16-bit opcode, obviously. That's why first 16-bit instruction loads higher 10 bits

[12] Branch with Link
[13] link register
[14] program counter
[15] Read more about this in next section 1.2
[16] MOVe
[17] Load Multiple Full Descending
[18] link register

of offset and second — loads 11 lower bits of offset. As I mentioned, all instructions in thumb mode has size of 2 bytes or 16 bits. This mean, it's not possible for thumb-instruction to be on odd address whatsoever. Considering this, last address bit may be omitted while instruction encoding. Summarizing, in BL thumb-instruction, $\pm \approx 2M$ can be encoded as offset from current address.

Other instructions in functions are: PUSH and POP works just like described STMFD/LDMFD, but SP register not mentioned explicitely here. ADR works just like in previous example. MOVS writes 0 in R0 register to zero returning.

Optimizing Xcode (LLVM) + ARM mode

Xcode 4.6.3 without optimization turned on, produces a lot of redundant code, so we'll study that version where instruction count as small as possible: -O3.

Listing 1.5: Optimizing Xcode (LLVM) + ARM mode

```
__text:000028C4             _hello_world
__text:000028C4 80 40 2D E9                 STMFD           SP!, {R7,LR}
__text:000028C8 86 06 01 E3                 MOV             R0, #0x1686
__text:000028CC 0D 70 A0 E1                 MOV             R7, SP
__text:000028D0 00 00 40 E3                 MOVT            R0, #0
__text:000028D4 00 00 8F E0                 ADD             R0, PC, R0
__text:000028D8 C3 05 00 EB                 BL              _puts
__text:000028DC 00 00 A0 E3                 MOV             R0, #0
__text:000028E0 80 80 BD E8                 LDMFD           SP!, {R7,PC}

__cstring:00003F62 48 65 6C 6C+aHelloWorld_0   DCB "Hello world!",0
```

Instructions STMFD and LDMFD are familiar to us.

MOV instruction just writes $0x1686$ number into R0 register, this is offset pointing to the "Hello world!" string.

R7 register, as it is standardized in [App10] is frame pointer, more on it below.

MOVT R0, #0 instruction writes 0 into higher 16 bit of register. The issue is here in that generic MOV instruction in ARM mode may writes only lower 16 bit of register. Remember, all instruction's opcodes in ARM mode are limited in size to 32 bits. Of course, this limitation is not related to moving between registers. So that's why additional instruction MOVT exist for writing into higher bits (from 16 to 31 inclusive). However, its usage here is redundant, because ''MOV R0, #0x1686'' instruction above cleared higher part of register. Probably, it's compiler's shortcoming.

''ADD R0, PC, R0'' instruction adding PC to R0, for calculating absolute address of "Hello world!" string, and as we already know that, it's "position-independent code", so this corrective is essential here.

BL instruction calling puts() instead of printf().

GCC replaced first printf() call to puts(). Indeed: printf() with sole argument is almost analogous to puts().

Almost, because we need to be sure that this string will not contain printf-control statements starting with %: then effect of these two functions will be different.

Why compiler replaced printf() to puts()? Because puts() () work faster [19].

puts() working faster because it just passes characters to stdout not comparing each with % symbol.

Next, we see familiar to us ''MOV R0, #0''instruction, intended to set 0 to R0 register.

Optimizing Xcode (LLVM) + thumb-2 mode

By default, Xcode 4.6.3 generating code for thumb-2 in such manner:

Listing 1.6: Optimizing Xcode (LLVM) + thumb-2 mode

```
__text:00002B6C                 _hello_world
__text:00002B6C 80 B5                       PUSH            {R7,LR}
__text:00002B6E 41 F2 D8 30                 MOVW            R0, #0x13D8
__text:00002B72 6F 46                       MOV             R7, SP
__text:00002B74 C0 F2 00 00                 MOVT.W          R0, #0
__text:00002B78 78 44                       ADD             R0, PC
```

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

```
__text:00002B7A 01 F0 38 EA                         BLX             _puts
__text:00002B7E 00 20                               MOVS            R0, #0
__text:00002B80 80 BD                               POP             {R7,PC}

...

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

BL and BLX instructions in thumb mode, as we remember, encoded as pair of 16-bit instructions and in thumb-2, these *surrogate* opcodes extended in such way so that new instruction may be encoded here as 32-bit instructions. That's easily observable — opcodes of thumb-2 instructions are also beginning with $0xFx$ or $0xEx$. But in IDA 5 listings, first byte of opcode is at the place of second, that's because instructions here encoded as follows: last byte and then first one (for thumb and thumb-2 modes), or, (for instructions in ARM mode): fourth byte, then third, then second and first. So as we see, MOVW, MOVT.W and BLX instructions are beginning with $0xFx$.

One of thumb-2 instructions is ‘‘MOVW R0, #0x13D8’’ — it writes 16-bit value into lower part of R0 register.

Also ‘‘MOVT.W R0, #0’’ — this instruction works just like MOVT from previous example, but it works in thumb-2.

Among other differences, here is BLX instruction used instead of BL. Difference in that way that beside saving of return address in LR register and passing control to puts() function, processor is switching from thumb mode to ARM or back. This instruction in place here because the instruction to which control is passed looks like (it's encoded in ARM mode):

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

So, observant reader may ask: why not to call puts() right at the place of code where it needed?

But that's not very space-efficient, and that's why.

Almost any program uses external dynamic libraries, like DLL in Windows, .so in *NIX or .dylib in Mac OS X. Often used library functions are stored in dynamic libraries, including standard C-function puts().

In executable binary file (Windows PE .exe, ELF or Mach-O) a section of imports is present, that is list of symbols (functions or global variables) being imported from external modules and also names of these modules.

Operation system loader loads all modules need and, while enumerating importing symbols in primary module, sets correct addresses of each symbol.

In our case, *__imp__puts* is 32-bit variable where OS loader will write correct address of that function in external library. So that LDR instruction just takes 32-bit value from this variable and, writing it into PC register, just passing control to it.

So to readuce a time OS loader needs for doing this procedure, it's good idea for it to write address of each symbol only once, to special place for it.

Besides, as we already figured out, it's not possible to load 32-bit value into register using only one instruction, without memory access. So, it is optimal to allocate separate function working in ARM mode with only one goal — to pass control to dynamic library. And then to jump to this short one-instruction function (so called thunk-function) from thumb-code.

By the way, in previous example (compiled for ARM mode) control passing by BL instruction is going to the same thunk-function, however, processor mode is not switched (hence, absence of “X” in instruction mnemonic).

1.2 Stack

Stack — is one of the most fundamental things in computer science.[20].

Technically, it's just a memory block in process memory + ESP or RSP register in x86, or SP register in ARM, as a pointer within this block.

Most frequently used stack access instructions are PUSH and POP (both in x86 and ARM thumb-mode). PUSH subtracting ESP/RSP/SP by 4 and then writing contents of its sole operand to the memory address pointing by ESP/RSP/SP.

POP is reverse operation: get a data from memory pointing by ESP/RSP/SP, put it to operand (often register) and then add 4 to ESP/RSP/SP. Of course, this is for 32-bit environment. 8 will be here instead of 4 in x64 environment.

After stack allocation, stack pointer pointing to the end of stack. PUSH increasing stack pointer, and POP decreasing. The end of stack is actually at the beginning of allocated for stack memory block. It seems strange, but it is so.

Nevertheless, ARM has instructions supporting ascending stacks, but also descending stacks. For example, STMFD[21]/LDMFD[22], STMED[23]/LDMED[24] instructions are intended for work with descending stack. STMFA[25]/LMDFA[26], STMEA[27]/LDMEA[28] instructions are intended for work with ascending stack.

What stack is used for?

1.2.1 Save return address where function should return control after execution

x86

While calling another function by CALL instruction, the address of point exactly after CALL instruction is saved to stack, and then unconditional jump to the address from CALL operand is executed.

CALL is PUSH address_after_call / JMP operand instructions pair equivalent.

RET is fetching value from stack and jump to it — it is POP tmp / JMP tmp instructions pair equivalent.

Stack overflow is simple, just run eternal recursion:

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

MSVC 2008 reporting about problem:

```
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
```

...but generates right code anyway:

```
?f@@YAXXZ PROC                                          ; f
; File c:\tmp6\ss.cpp
; Line 2
        push    ebp
        mov     ebp, esp
; Line 3
        call    ?f@@YAXXZ                               ; f
```

[20] http://en.wikipedia.org/wiki/Call_stack
[21] Store Multiple Full Descending
[22] Load Multiple Full Descending
[23] Store Multiple Empty Descending
[24] Load Multiple Empty Descending
[25] Store Multiple Full Ascending
[26] Load Multiple Full Ascending
[27] Store Multiple Empty Ascending
[28] Load Multiple Empty Ascending

```
; Line 4
        pop     ebp
        ret     0
?f@@YAXXZ ENDP                                          ; f
```

...Also, if we turn on optimization (`/Ox` option), the optimized code will not overflow stack, but will work *correctly*[29]:

```
?f@@YAXXZ PROC                                          ; f
; File c:\tmp6\ss.cpp
; Line 2
$LL3@f:
; Line 3
        jmp     SHORT $LL3@f
?f@@YAXXZ ENDP                                          ; f
```

GCC 4.4.1 generating the same code in both cases, although not warning about problem.

ARM

ARM programs also use stack for return address savings, but differently. As it was mentioned in "Hello, world!" 1.1.2, return address is saved to `LR` (*link register*). However, if one need to call some another function and use `LR` register one more time, its value should be saved. Usually, it is saved in function prologue, often, we see there instruction like ''`PUSH R4-R7,LR`'', and also this instruction in epilogue ''`POP R4-R7,PC`'' — thus register values to be used in function are saved in stack, including `LR`.

Nevertheless, if some function never call any other function, in ARM terminology it's called *leaf function*[30]. As a consequence, "leaf" function don't use `LR` register. And if this function is small, if it use small number of registers, it may not use stack at all. Thus, it is possible to call "leaf" function without stack usage. This could be faster than in x86, because external RAM is not used for stack [31] . Or, it can be useful for such situations, when memory for stack is not yet allocated or not available.

1.2.2 Function arguments passing

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

Callee[32] function get its arguments via stack ponter.

See also section about calling conventions 2.5.

It is important to note that no one oblige programmers to pass arguments through stack, it is not prerequisite.

One could implement any other method not using stack.

For example, it is possible to allocate a place for arguments in heap, fill it and pass to a function via pointer to this pack in `EAX` register. And this will work [33].

However, it is convenient tradition in x86 and ARM to use stack for this.

1.2.3 Local variable storage

A function could allocate some space in stack for its local variables just shifting stack pointer deeply enough to stack bottom.

It is also not prerequisite. You could store local variables wherever you like. But traditionally it is so.

[29] irony here

[30] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html

[31] Some time ago, on PDP-11 and VAX, CALL instruction (calling other functions) was expensive, up to 50% of execution time might be spent on it, so it was common sense that big number of small function is anti-pattern [Ray03, Chapter 4, Part II].

[32] Function being called

[33] For example, in "The Art of Computer Programming" book by Donald Knuth, in section 1.4.1 dedicated to subroutines [Knu98, section 1.4.1], we can read about one way to supply arguments to subroutine is simply to list them after the `JMP` instruction passing control to subroutine. Knuth writes that this method was particularly convenient on System/360.

x86: alloca() function

It is worth noting `alloca()` function.[34].

This function works like `malloc()`, but allocate memory just in stack.

Allocated memory chunk is not needed to be freed via `free()` function call because function epilogue 2.2 will return ESP back to initial state and allocated memory will be just annuled.

It is worth noting how `alloca()` implemented.

This function, if to simplify, just shifting ESP deeply to stack bottom so much bytes you need and set ESP as a pointer to that *allocated* block. Let's try:

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

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

    puts (buf);
};
```

(`_snprintf()` function works just like `printf()`, but instead dumping result into stdout (e.g., to terminal or console), write it to `buf` buffer. `puts()` copies `buf` contents to stdout. Of course, these two function calls might be replaced by one `printf()` call, but I would like to illustrate small buffer usage.)

Let's compile (MSVC 2010):

Listing 1.7: MSVC 2010

```
...

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

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

    push   esi
    call   _puts
    add    esp, 28          ; 0000001cH

...
```

The sole `alloca()` argument passed via EAX (instead of pushing into stack) [35]. After `alloca()` call, ESP is now pointing to the block of 600 bytes and we can use it as memory for `buf` array.

GCC 4.4.1 can do the same without calling external functions:

Listing 1.8: GCC 4.4.1

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

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

                push    ebp
```

[34] As of MSVC, function implementation can be found in `alloca16.asm` and `chkstk.asm` in `C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\crt\src\intel`

[35] It's because alloca() is rather compiler intrinsic than usual function

```
                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          ; align pointer
                shl     eax, 4          ; by 16-byte border
                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
```

1.2.4 (Windows) SEH

SEH (*Structured Exception Handling*) records are also stored in stack (if needed). [36].

1.2.5 Buffer overflow protection

More about it here 1.14.2.

[36] Classic Matt Pietrek article about SEH: http://www.microsoft.com/msj/0197/Exception/Exception.aspx

1.3 `printf()` with several arguments

Now let's extend *Hello, world!* 1.1 example, replacing `printf()` in `main()` function body by this:

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

1.3.1 x86

Let's compile it by MSVC 2010 and we got:

```
$SG3830 DB      'a=%d; b=%d; c=%d', 00H

...

        push    3
        push    2
        push    1
        push    OFFSET $SG3830
        call    _printf
        add     esp, 16                                 ; 00000010H
```

Almost the same, but now we can see that `printf()` arguments are pushing into stack in reverse order: and the first argument is pushing in as the last one.

By the way, variables of *int* type in 32-bit environment has 32-bit width, that's 4 bytes.

So, we got here 4 arguments. $4 * 4 = 16$ — they occupy in stack exactly 16 bytes: 32-bit pointer to string and 3 number of *int* type.

When stack pointer (ESP register) is corrected by ''`ADD ESP, X`'' instruction after some function call, often, the number of function arguments could be deduced here: just divide X by 4.

Of course, this is related only to *cdecl* calling convention.

See also section about calling conventions 2.5.

It is also possible for compiler to merge several ''`ADD ESP, X`'' instructions into one, after last call:

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

Now let's compile the same in Linux by GCC 4.4.1 and take a look in IDA 5 what we got:

```
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
```

It can be said, the difference between code by MSVC and GCC is only in method of placing arguments into stack. Here GCC working directly with stack without PUSH/POP.

1.3.2 ARM: 3 printf() arguments

Traditionally, ARM arguments passing scheme (calling convention) is as follows: the 4 first arguments are passed in R0-R3 registers and remaining arguments — via stack. This resembling arguments passing scheme in fastcall 2.5.3 or win64 2.5.5.

Non-optimizing Keil + ARM mode

Listing 1.9: Non-optimizing Keil + ARM mode

```
.text:00000014             printf_main1
.text:00000014 10 40 2D E9                 STMFD   SP!, {R4,LR}
.text:00000018 03 30 A0 E3                 MOV     R3, #3
.text:0000001C 02 20 A0 E3                 MOV     R2, #2
.text:00000020 01 10 A0 E3                 MOV     R1, #1
.text:00000024 1D 0E 8F E2                 ADR     R0, aADBDCD     ; "a=%d; b=%d; c=%d\n"
.text:00000028 0D 19 00 EB                 BL      __2printf
.text:0000002C 10 80 BD E8                 LDMFD   SP!, {R4,PC}
```

So, the first 4 arguments are passing via R0-R0 arguments in this order: pointer to printf() format string in R0, then 1 in R1, 2 in R2 and 3 in R3.

Nothing unusual so far.

Optimizing Keil + ARM mode

Listing 1.10: Optimizing Keil + ARM mode

```
.text:00000014                             EXPORT printf_main1
.text:00000014             printf_main1
.text:00000014 03 30 A0 E3                 MOV     R3, #3
.text:00000018 02 20 A0 E3                 MOV     R2, #2
.text:0000001C 01 10 A0 E3                 MOV     R1, #1
.text:00000020 1E 0E 8F E2                 ADR     R0, aADBDCD     ; "a=%d; b=%d; c=%d\n"
.text:00000024 CB 18 00 EA                 B       __2printf
```

This is optimized (-O3) version for ARM mode and here we see B as the last instruction instead of familiar BL. Another difference between this optimized version and previous one, compiled without optimization, is also in the fact that there are no function prologue and epilogue (instructions saving R0 and LR registers values). B instruction just jumping to another address, without any LR register manipulation, that is, it's JMP analogue in x86. Why it works fine? Because this code is in fact equivalent to the previous. There are two main reasons: 1) stack is not modified, as well as SP stack pointer; 2) printf() call is the last one, there are nothing going on after it. After finishing, printf() function will just return control to the address stored in LR. But the address of the place from where our function was called is now in LR! And consequently, control from printf() will returned to that place. As a consequent, we don't need to save LR, because we don't need to modify LR. And we don't need to modify LR because there are no other functions calls except printf(), furthermore, after this call we are not planning to do anything! That's why this optimization is possible.

Another similar example was described in "switch()/case/default" section, here 1.9.1.

Optimizing Keil + thumb mode

Listing 1.11: Optimizing Keil + thumb mode

```
.text:0000000C             printf_main1
.text:0000000C 10 B5                       PUSH    {R4,LR}
.text:0000000E 03 23                       MOVS    R3, #3
.text:00000010 02 22                       MOVS    R2, #2
.text:00000012 01 21                       MOVS    R1, #1
.text:00000014 A4 A0                       ADR     R0, aADBDCD     ; "a=%d; b=%d; c=%d\n"
.text:00000016 06 F0 EB F8                 BL      __2printf
.text:0000001A 10 BD                       POP     {R4,PC}
```

There are no significant difference from non-optimized code for ARM mode.

1.3.3 ARM: 8 `printf()` arguments

To see, how other arguments will be passed via stack, let's change our example again by increasing number of arguments to be passed to 9 (`printf()` format string + 8 *int* variables):

```
void printf_main2()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
};
```

Optimizing Keil: ARM mode

```
.text:00000028             printf_main2
.text:00000028
.text:00000028             var_18          = -0x18
.text:00000028             var_14          = -0x14
.text:00000028             var_4           = -4
.text:00000028
.text:00000028 04 E0 2D E5                 STR     LR, [SP,#var_4]!
.text:0000002C 14 D0 4D E2                 SUB     SP, SP, #0x14
.text:00000030 08 30 A0 E3                 MOV     R3, #8
.text:00000034 07 20 A0 E3                 MOV     R2, #7
.text:00000038 06 10 A0 E3                 MOV     R1, #6
.text:0000003C 05 00 A0 E3                 MOV     R0, #5
.text:00000040 04 C0 8D E2                 ADD     R12, SP, #0x18+var_14
.text:00000044 0F 00 8C E8                 STMIA   R12, {R0-R3}
.text:00000048 04 00 A0 E3                 MOV     R0, #4
.text:0000004C 00 00 8D E5                 STR     R0, [SP,#0x18+var_18]
.text:00000050 03 30 A0 E3                 MOV     R3, #3
.text:00000054 02 20 A0 E3                 MOV     R2, #2
.text:00000058 01 10 A0 E3                 MOV     R1, #1
.text:0000005C 6E 0F 8F E2                 ADR     R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d
    ; g=%"...
.text:00000060 BC 18 00 EB                 BL      __2printf
.text:00000064 14 D0 8D E2                 ADD     SP, SP, #0x14
.text:00000068 04 F0 9D E4                 LDR     PC, [SP+4+var_4],#4
```

This code can be divided into several parts:

- Function prologue:

  The very first "`STR LR, [SP,#var_4]!`" instruction save `LR` in stack, because we will use this register for `printf()` call.

  The second "`SUB SP, SP, #0x14`" instruction decreasing `SP` stack pointer, but in fact, this procedure is needed for allocating a space of size $0x14$ (20) bytes in the stack. Indeed, we need to pass 5 32-bit values via stack to `printf()`, and each one occupy 4 bytes, that is $5 * 4 = 20$ — exactly. Other 4 32-bit values will be passed in registers.

- Passing 5, 6, 7 and 8 via stack:

  Then values 5, 6, 7 and 8 are written to R0, R1, R2 and R3 registers respectively. Then ‘‘ADD R12, SP, #0x18+var_14’’ instruction write an address of place in stack, where these 4 variables will be written, into R12 register. *var_14* is an assembly macro, equal to $-0x14$, such macros are created by IDA 5 in order to show simply how code accessing stack. *var_?* macros created by IDA 5 reflecting local variables in stack. So, $SP + 4$ will be written into R12 register. The next ‘‘STMIA R12, R0-R3’’ instruction writes R0-R3 registers contents at the place in memory to which R12 pointing. STMIA instruction meaning *Store Multiple Increment After*. *Increment After* meaning that R12 will be increasing by 4 after each register value write.

- Passing 4 via stack: 4 is stored in R0 and then, this value, with the help of ‘‘STR R0, [SP,#0x18+var_18]’’ instruction, is saved in stack. *var_18* is $-0x18$, offset will be 0, so, value from R0 register (4) will be written to a place SP pointing to.

- Passing 1, 2 and 3 via registers:

  Values of first 3 numbers (a, b, c) (1, 2, 3 respectively) are passing in R1, R2 and R3 registers right before printf() call, and other 5 values are passed via stack and this is how:

- printf() call:

- Function epilogue:

  ‘‘ADD SP, SP, #0x14’’ instruction returning SP to former place, thus annulling what was written to stack. Of course, what was written in stack will stay there, but it all will be rewritten while execution of following functions.

  ‘‘LDR PC, [SP+4+var_4],#4’’ instruction loading saved in stack LR value into PC, providing exit from the function.

Optimizing Keil: thumb mode

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

Almost same as in previous example, however, this is thumb code and values are packed into stack differently: 8 for the first time, then 5, 6, 7 for the second and 4 for the third.

Optimizing Xcode (LLVM): ARM mode

```
__text:0000290C             _printf_main2
__text:0000290C
__text:0000290C             var_1C          = -0x1C
__text:0000290C             var_C           = -0xC
__text:0000290C
__text:0000290C 80 40 2D E9                 STMFD           SP!, {R7,LR}
__text:00002910 0D 70 A0 E1                 MOV             R7, SP
__text:00002914 14 D0 4D E2                 SUB             SP, SP, #0x14
__text:00002918 70 05 01 E3                 MOV             R0, #0x1570
__text:0000291C 07 C0 A0 E3                 MOV             R12, #7
__text:00002920 00 00 40 E3                 MOVT            R0, #0
__text:00002924 04 20 A0 E3                 MOV             R2, #4
__text:00002928 00 00 8F E0                 ADD             R0, PC, R0
__text:0000292C 06 30 A0 E3                 MOV             R3, #6
__text:00002930 05 10 A0 E3                 MOV             R1, #5
__text:00002934 00 20 8D E5                 STR             R2, [SP,#0x1C+var_1C]
__text:00002938 0A 10 8D E9                 STMFA           SP, {R1,R3,R12}
__text:0000293C 08 90 A0 E3                 MOV             R9, #8
__text:00002940 01 10 A0 E3                 MOV             R1, #1
__text:00002944 02 20 A0 E3                 MOV             R2, #2
__text:00002948 03 30 A0 E3                 MOV             R3, #3
__text:0000294C 10 90 8D E5                 STR             R9, [SP,#0x1C+var_C]
__text:00002950 A4 05 00 EB                 BL              _printf
__text:00002954 07 D0 A0 E1                 MOV             SP, R7
__text:00002958 80 80 BD E8                 LDMFD           SP!, {R7,PC}
```

Almost the same what we already figured out, with the exception of `STMFA` (Store Multiple Full Ascending) instruction, it is synonym to `STMIB` (Store Multiple Increment Before) instruction. This instruction increasing `SP` and only then writing next register value into memory, but not vice versa.

Another thing we easily spot is that instructions ostensibly located randomly. For instance, `R0` register value is prepared in three places, at addresses $0x2918$, $0x2920$ and $0x2928$, when it would be possible to do it in one single place. However, optimizing compiler has his own reasons about how to place instructions better. Usually, processor attempts to execute instructions located side-by-side. For example, instructions like ‘‘`MOVT R0, #0`’’ and ‘‘`ADD R0, PC, R0`’’ cannot be executed simultaneously, because they both modifying `R0` register. On the other hand, ‘‘`MOVT R0, #0`’’ and ‘‘`MOV R2, #4`’’ instructions can be executed simultaneously because effects of their execution are not conflicting with each other. Presumably, compiler tries to generate code in such way, where it’s possible, of course.

Optimizing Xcode (LLVM): thumb-2 mode

```
__text:00002BA0             _printf_main2
__text:00002BA0
__text:00002BA0             var_1C          = -0x1C
__text:00002BA0             var_18          = -0x18
__text:00002BA0             var_C           = -0xC
__text:00002BA0
__text:00002BA0 80 B5                       PUSH            {R7,LR}
__text:00002BA2 6F 46                       MOV             R7, SP
__text:00002BA4 85 B0                       SUB             SP, SP, #0x14
__text:00002BA6 41 F2 D8 20                 MOVW            R0, #0x12D8
__text:00002BAA 4F F0 07 0C                 MOV.W           R12, #7
__text:00002BAE C0 F2 00 00                 MOVT.W          R0, #0
__text:00002BB2 04 22                       MOVS            R2, #4
__text:00002BB4 78 44                       ADD             R0, PC  ; char *
__text:00002BB6 06 23                       MOVS            R3, #6
__text:00002BB8 05 21                       MOVS            R1, #5
__text:00002BBA 0D F1 04 0E                 ADD.W           LR, SP, #0x1C+var_18
__text:00002BBE 00 92                       STR             R2, [SP,#0x1C+var_1C]
__text:00002BC0 4F F0 08 09                 MOV.W           R9, #8
__text:00002BC4 8E E8 0A 10                 STMIA.W         LR, {R1,R3,R12}
```

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

Almost the same as in previous example, with the exception that thumb-instructions are used there instead.

1.3.4 By the way

By the way, this difference between passing arguments in x86 and ARM is a good illustration that CPU is not aware of how arguments is passed to functions. It is also possible to create hypothetical compiler which is able to pass arguments via some special structure not using stack at all.

1.4 scanf()

Now let's use scanf().

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

        scanf ("%d", &x);

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

        return 0;
};
```

OK, I agree, it is not clever to use `scanf()` today. But I wanted to illustrate passing pointer to *int*.

1.4.1 About pointers

It's one of the most fundamental things in computer science. Often, large array, structure or object, it's too costly to pass to another function, while passing its address is much easier. More than that: if calling function should modify something in that large array or structure, to return it as a whole is absurdical as well. So the most simple thing to do is to pass an address of array or structure to function, and let it change it what need.

In C/C++ it's just an address of some place in memory.

In x86, address is represented as 32-bit number (i.e., occupying 4 bytes), while in x86-64 it's 64-bit number (occupying 8 bytes). By the way, that's a reson of some people's indignation related to switching to x86-64 — all pointers on that architecture will require twice as more space.

With some effort, it's possible to work only with untyped pointers, for example, standard function `memcpy()`, copying a block from one place in memory to another, takes 2 pointers of `void*` type on input, because it's not possible to predict block type you would like to copy, and it's not even important to know, only block size is important.

Also, pointers are widely used when function need to return more than one value (we will back to this in future 1.7). *scanf()* is just that case. In addition to the function's need to show, how many values were read successfully, it also should to return all these values.

In C/C++ pointer type is needed only for type checking on compiling stage. Internally, in compiled code, there are no information about pointers types.

1.4.2 x86

What we got after compiling in 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
```

Variable x is local.

C/C++ standard tell us it must be visible only in this function and not from any other place. Traditionally, local variables are placed in the stack. Probably, there could be other ways, but in x86 it is so.

Next after function prologue instruction PUSH ECX is not for saving ECX state (notice absence of corresponding POP ECX at the function end).

In fact, this instruction just allocate 4 bytes in stack for x variable storage.

x will be accessed with the assistance of _x$ macro (it equals to -4) and EBP register pointing to current frame.

Over a span of function execution, EBP is pointing to current stack frame and it is possible to have an access to local variables and function arguments via EBP+offset.

It is also possible to use ESP, but it's often changing and not very convenient. So it can be said, EBP is *frozen state* of ESP at the moment of function execution start.

Function scanf() in our example has two arguments.

First is pointer to the string containing ''%d'' and second — address of variable x.

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

LEA meaning *load effective address*, but over a time it changed its primary application 2.1.

It can be said, LEA here just placing to EAX sum of EBP value and _x$ macro.

It is the same as lea eax, [ebp-4].

So, 4 subtracting from EBP value and result is placed to EAX. And then value in EAX is pushing into stack and scanf() is called.

After that, printf() is called. First argument is pointer to string: ''You entered %d...\n''.

Second argument is prepared as: mov ecx, [ebp-4], this instruction placing to ECX not address of x variable but its contents.

After, ECX value is placing into stack and last printf() called.

Let's try to compile this code in GCC 4.4.1 under Linux:

```
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 replaced first `printf()` call to `puts()`, it was already described 1.1.2 why it was done.
As before — arguments are placed into stack by `MOV` instruction.

1.4.3 ARM

Optimizing Keil + thumb mode

```
.text:00000042             scanf_main
.text:00000042
.text:00000042             var_8           = -8
.text:00000042
.text:00000042 08 B5                       PUSH    {R3,LR}
.text:00000044 A9 A0                       ADR     R0, aEnterX     ; "Enter X:\n"
.text:00000046 06 F0 D3 F8                 BL      __2printf
.text:0000004A 69 46                       MOV     R1, SP
.text:0000004C AA A0                       ADR     R0, aD          ; "%d"
.text:0000004E 06 F0 CD F8                 BL      __0scanf
.text:00000052 00 99                       LDR     R1, [SP,#8+var_8]
.text:00000054 A9 A0                       ADR     R0, aYouEnteredD___ ; "You entered %d...\n"
.text:00000056 06 F0 CB F8                 BL      __2printf
.text:0000005A 00 20                       MOVS    R0, #0
.text:0000005C 08 BD                       POP     {R3,PC}
```

A pointer to *int*-typed variable shoud be passed to `scanf()` so it can return value via it. *int* is 32-bit value, so we need 4 bytes for storing it somewhere in memory, and it fits exactly in 32-bit regsiter. A place for local variable `x` is allocated in stack and IDA 5 named it *var_8*, however, it's not necessary to allocate it, because, `SP` stack pointer is already pointing to the place which may be used. So, `SP` stack pointer value is copied to `R1` register and, together with format-string, passed into `scanf()`. Later, with the help of `LDR` instruction, this value is moved from stack into `R1` register in order to be passed into `printf()`.

Examples compiled for ARM-mode and also examples compiled with Xcode LLVM are not differ significally from what we saw here, so they are omitted.

1.4.4 Global variables

x86

What if `x` variable from previous example will not be local but global variable? Then it will be accessible from any place but not only from function body. It is not very good programming practice, but for the sake of experiment we could do this.

```
_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
```

Now `x` variable is defined in `_DATA` segment. Memory in local stack is not allocated anymore. All accesses to it are not via stack but directly to process memory. Its value is not defined. This mean that memory will be allocated by operation system, but not compiler, neither operation system will not take care about its initial value at the moment of `main()` function start. As experiment, try to declare large array and see what will it contain after program loading.

Now let's assign value to variable explicitly:

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

We got:

```
_DATA   SEGMENT
_x      DD      0aH

...
```

Here we see value 0xA of DWORD type (DD meaning DWORD = 32 bit).

If you will open compiled .exe in IDA 5, you will see *x* placed at the beginning of `_DATA` segment, and after you'll see text strings.

If you will open compiled .exe in IDA 5 from previous example where *x* value is not defined, you'll see something like this:

```
.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` marked as `?` among another variables not required to be initialized. This mean that after loading .exe to memory, place for all these variables will be allocated and some random garbage will be here. But in .exe file these not initialized variables are not occupy anything. It is suitable for large arrays, for example.

It is almost the same in Linux, except segment names and properties: not initialized variables are located in `_bss` segment. In ELF[37] file format this segment has such attributes:

[37] Executable file format widely used in *NIX system including Linux

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

If to assign some value to variable, e.g. 10, it will be placed in `_data` segment, this is segment with such attributes:

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

ARM: Optimizing Keil + thumb mode

```
.text:00000000 ; Segment type: Pure code
.text:00000000                 AREA .text, CODE
...
.text:00000000 main
.text:00000000                 PUSH    {R4,LR}
.text:00000002                 ADR     R0, aEnterX     ; "Enter X:\n"
.text:00000004                 BL      __2printf
.text:00000008                 LDR     R1, =x
.text:0000000A                 ADR     R0, aD          ; "%d"
.text:0000000C                 BL      __0scanf
.text:00000010                 LDR     R0, =x
.text:00000012                 LDR     R1, [R0]
.text:00000014                 ADR     R0, aYouEnteredD___ ; "You entered %d...\n"
.text:00000016                 BL      __2printf
.text:0000001A                 MOVS    R0, #0
.text:0000001C                 POP     {R4,PC}
...
.text:00000020 aEnterX         DCB "Enter X:",0xA,0    ; DATA XREF: main+2
.text:0000002A                 DCB    0
.text:0000002B                 DCB    0
.text:0000002C off_2C          DCD x                   ; DATA XREF: main+8
.text:0000002C                                         ; main+10
.text:00000030 aD              DCB "%d",0              ; DATA XREF: main+A
.text:00000033                 DCB    0
.text:00000034 aYouEnteredD___ DCB "You entered %d...",0xA,0 ; DATA XREF: main+14
.text:00000047                 DCB 0
.text:00000047 ; .text         ends
.text:00000047
...
.data:00000048 ; Segment type: Pure data
.data:00000048                 AREA .data, DATA
.data:00000048                 ; ORG 0x48
.data:00000048                 EXPORT x
.data:00000048 x               DCD 0xA                 ; DATA XREF: main+8
.data:00000048                                         ; main+10
.data:00000048 ; .data         ends
```

So, `x` variable is now global and it's localted, and, somehow, it's now located in other segment, namely data segment (*.data*). One could ask, why text strings are located in code segment (*.text*) and `x` can be located right here? Because, this is variable, and by its definition, it can be changed. And probably, can be changed very often. Segment of code not infrequently can be located in microcontroller ROM (remember, we now deal with embedded microelectronics, and memory scarcity is common here), and changeable variables — in RAM. It's not very economically to store constant variables in RAM when one have ROM.

Onwards, we see, in code segment, a pointer to `x` (`off_2C`) variable, and all operations with variable occured via this pointer. This is because `x` variable can be located somewhere far from this code fragment, so its address should be saved somewhere in close proximity to the code. `LDR` instruction in thumb mode can address only variable in range of 1020 bytes from the place where it's located. Same instruction in ARM-mode — variables in range ± 4095 bytes, this, address of `x` variable should be located somewhere in close proximity, because, there are no guarantee that linker will able to place this variable near the code, it could be even in other memory chip!

One more thing: if variable will be declared as *const*, Keil compiler will allocate it in `.constdata` segment. Perhaps, thereafter, linker will able to place this segment in ROM too, along with code segment.

1.4.5 scanf() result checking

x86

As I noticed before, it is slightly old-fashioned to use `scanf()` today. But if we have to, we need at least check if `scanf()` finished correctly without error.

```
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;
};
```

By standard, `scanf()`[38] function returning number of fields it successfully read.

In our case, if everything went fine and user entered some number, `scanf()` will return 1 or 0 or EOF in case of error.

I added C code for `scanf()` result checking and printing error message in case of error.

What we got in assembly language (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 $LN2@main
; Line 9
        mov     ecx, DWORD PTR _x$[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
```

Caller function (`main()`) should have access to result of callee function (`scanf()`), so callee leave this value in `EAX` register.

After, we check it using instruction `CMP EAX, 1` (*CoMPare*), in other words, we compare value in `EAX` with 1.

`JNE` conditional jump follows `CMP` instruction. `JNE` mean *Jump if Not Equal*.

So, if `EAX` value not equals to 1, then the processor will pass execution to the address mentioned in operand of `JNE`, in our case it is `$LN2@main`. Passing control to this address, microprocesor will execute function `printf()` with argument "`What you entered? Huh?`". But if everything is fine, conditional jump will not be taken, and another `printf()` call will be executed, with two arguments: '`You entered %d...`' and value of variable `x`.

[38] MSDN: scanf, wscanf

Because second subsequent `printf()` not needed to be executed, there are `JMP` after (unconditional jump) it will pass control to the place after second `printf()` and before `XOR EAX, EAX` instruction, which implement `return 0`.

So, it can be said that most often, comparing some value with another is implemented by `CMP`/`Jcc` instructions pair, where *cc* is *condition code*. `CMP` comparing two values and set processor flags[39]. `Jcc` check flags needed to be checked and pass control to mentioned address (or not pass).

But in fact, this could be perceived paradoxial, but `CMP` instruction is in fact `SUB` (subtract). All arithmetic instructions set processor flags too, not only `CMP`. If we compare 1 and 1, $1 - 1$ will be zero in result, `ZF` flag will be set (meaning that last result was zero). There are no any other circumstances when it's possible except when operands are equal. `JNE` checks only `ZF` flag and jumping only if it is not set. `JNE` is in fact a synonym of `JNZ` (*Jump if Not Zero*) instruction. Assembler translating both `JNE` and `JNZ` instructions into one single opcode. So, `CMP` can be replaced to `SUB` and almost everything will be fine, but the difference is in that `SUB` alter value at first operand. `CMP` is *"SUB without saving result"*.

Code generated by GCC 4.4.1 in Linux is almost the same, except differences we already considered.

ARM: Optimizing Keil + thumb mode

Listing 1.12: Optimizing Keil + thumb mode

```
var_8           = -8

                PUSH    {R3,LR}
                ADR     R0, aEnterX     ; "Enter X:\n"
                BL      __2printf
                MOV     R1, SP
                ADR     R0, aD          ; "%d"
                BL      __0scanf
                CMP     R0, #1
                BEQ     loc_1E
                ADR     R0, aWhatYouEntered ; "What you entered? Huh?\n"
                BL      __2printf

loc_1A                                  ; CODE XREF: main+26
                MOVS    R0, #0
                POP     {R3,PC}

loc_1E                                  ; CODE XREF: main+12
                LDR     R1, [SP,#8+var_8]
                ADR     R0, aYouEnteredD___ ; "You entered %d...\n"
                BL      __2printf
                B       loc_1A
```

New instructions here are `CMP` and `BEQ`.

`CMP` is similar to the x86 instruction, it subtracts one argument from another and save flags.

`BEQ` (*Branch Equal*) is jumping to another address if operands while comparing were equal to each other, or, if result of last computation was zero, or if Z flag is 1. Same thing as `JZ` in x86.

Everything else is simple: execution flow is forking into two branches, then the branches are converging at the place where 0 is written into `R0`, as a value returned from the function, and then function finishing.

[39] About x86 flags, see also: http://en.wikipedia.org/wiki/FLAGS_register_(computing).

1.5 Passing arguments via stack

Now we figured out that caller function passing arguments to callee via stack. But how callee[40] access them?

```
#include <stdio.h>

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

int main()
{
        printf ("%d\n", f(1, 2, 3));
        return 0;
};
```

1.5.1 x86

What we have after compilation (MSVC 2010 Express):

Listing 1.13: 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
```

What we see is that 3 numbers are pushing to stack in function `main()` and `f(int,int,int)` is called then. Argument access inside `f()` is organized with help of macros like: `_a$ = 8`, in the same way as local variables

[40] function being called

accessed, but difference in that these offsets are positive (addressed with *plus* sign). So, adding `_a$` macro to `EBP` register value, *outer* side of stack frame is addressed.

Then `a` value is stored into `EAX`. After `IMUL` instruction execution, `EAX` value is product[41] of `EAX` and what is stored in `_b`. After `IMUL` execution, `ADD` is summing `EAX` and what is stored in `_c`. Value in `EAX` is not needed to be moved: it is already in place it need. Now return to caller — it will take value from `EAX` and used it as `printf()` argument.

Let's compile the same in GCC 4.4.1:

Listing 1.14: GCC 4.4.1

```
                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
```

Almost the same result.

1.5.2 ARM

Non-optimizing Keil + ARM mode

```
.text:000000A4 00 30 A0 E1                MOV     R3, R0
.text:000000A8 93 21 20 E0                MLA     R0, R3, R1, R2
.text:000000AC 1E FF 2F E1                BX      LR
...
.text:000000B0              main
.text:000000B0 10 40 2D E9                STMFD   SP!, {R4,LR}
```

[41] result of multiplication

```
.text:000000B4 03 20 A0 E3                MOV     R2, #3
.text:000000B8 02 10 A0 E3                MOV     R1, #2
.text:000000BC 01 00 A0 E3                MOV     R0, #1
.text:000000C0 F7 FF FF EB                BL      f
.text:000000C4 00 40 A0 E1                MOV     R4, R0
.text:000000C8 04 10 A0 E1                MOV     R1, R4
.text:000000CC 5A 0F 8F E2                ADR     R0, aD_0        ; "%d\n"
.text:000000D0 E3 18 00 EB                BL      __2printf
.text:000000D4 00 00 A0 E3                MOV     R0, #0
.text:000000D8 10 80 BD E8                LDMFD   SP!, {R4,PC}
```

In `main()` function, two other functions are simply called, and three values are passed to the first one (`f`).

As I mentioned before, in ARM, first 4 values are usually passed in first 4 registers (`R0`-`R3`).

`f`function, as it seems, use first 3 registers (`R0`-`R2`) as arguments.

`MLA` (*Multiply Accumulate*) instruction multiplicates two first operands (`R3` and `R1`), adds third operand (`R2`) to product and places result into zeroth operand (`R0`), via which, by standard, values are returned from functions.

Multiplication and addition at once[42] (*Fused multiply–add*) is very useful operation, by the way, there are no such instruction in x86, if not to count new FMA-instruction[43] in SIMD.

The very first `MOV R3, R0`, instruction, apparently, redundant (single `MLA` instruction could be used here instead), compiler wasn't optimized it, because, this is non-optimizing compilation.

`BX` instruction returns control to the address stored in `LR` and, if need, switches processor mode from thumb to ARM or vice versa. This can be necessary because, as we can see, `f` function is not aware, from which code it may be called, from ARM or thumb. This, if it will be called from thumb code, `BX` will not only return control to the calling function, but also will switch processor mode to thumb mode. Or not switch, if the function was called from ARM code.

Optimizing Keil + ARM mode

```
.text:00000098                f
.text:00000098 91 20 20 E0                MLA     R0, R1, R0, R2
.text:0000009C 1E FF 2F E1                BX      LR
```

And here is `f` function compiled by Keil compiler in full optimization mode (`-O3`). `MOV` instruction was optimized and now `MLA` uses all input registers and place result into `R0`, exactly where calling function will read it and use.

Optimizing Keil + thumb mode

```
.text:0000005E 48 43                      MULS    R0, R1
.text:00000060 80 18                      ADDS    R0, R0, R2
.text:00000062 70 47                      BX      LR
```

`MLA` instruction is not available in thumb mode, so, compiler generates the code doing these two operations separately. First `MULS` instruction multiply `R0` by `R1` leaving result in `R1`. Second (`ADDS`) instruction adds result and `R2` leaving result in `R0`.

[42] wikipedia: Multiply–accumulate operation

[43] https://en.wikipedia.org/wiki/FMA_instruction_set

1.6 One more word about results returning.

As of x86, function execution result is usually returned[44] in EAX register. If it's byte type or character (*char*) — then in lowest register EAX part — AL. If function returning *float* number, FPU register ST(0) will be used instead. In ARM, result is usually returned in R0 register.

That is why old C compilers can't create functions capable of returning something not fitting in one register (usually type *int*), but if one need it, one should return information via pointers passed in function arguments. Now it is possible, to return, let's say, whole structure, but its still not very popular. If function should return a large structure, caller must allocate it and pass pointer to it via first argument, hiddenly and transparently for programmer. That is almost the same as to pass pointer in first argument manually, but compiler hide this.

Small example:

```
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;
};
```

...what we got (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
```

Macro name for internal variable passing pointer to structure is $T3853 here.

[44] See also: MSDN: Return Values (C++)

1.7 Pointers

Pointers are often used to return values from function (recall scanf() case 1.4). For example, when function should return two values:

```
void f1 (int x, int y, int *sum, int *product)
{
        *sum=x+y;
        *product=x*y;
};

void main()
{
        int sum, product;

        f1(123, 456, &sum, &product);
        printf ("sum=%d, product=%d\n", sum, product);
};
```

This compiling into:

Listing 1.15: Optimizing MSVC 2010

```
CONST   SEGMENT
$SG3863 DB      'sum=%d, product=%d', 0aH, 00H
$SG3864 DB      'sum=%d, product=%d', 0aH, 00H
CONST   ENDS
_TEXT   SEGMENT
_x$ = 8                                                 ; size = 4
_y$ = 12                                                ; size = 4
_sum$ = 16                                              ; size = 4
_product$ = 20                                          ; size = 4
f1 PROC                                 ; f1
        mov     ecx, DWORD PTR _y$[esp-4]
        mov     eax, DWORD PTR _x$[esp-4]
        lea     edx, DWORD PTR [eax+ecx]
        imul    eax, ecx
        mov     ecx, DWORD PTR _product$[esp-4]
        push    esi
        mov     esi, DWORD PTR _sum$[esp]
        mov     DWORD PTR [esi], edx
        mov     DWORD PTR [ecx], eax
        pop     esi
        ret     0
f1 ENDP                                 ; f1

_product$ = -8                                          ; size = 4
_sum$ = -4                                              ; size = 4
_main   PROC
        sub     esp, 8
        lea     eax, DWORD PTR _product$[esp+8]
        push    eax
        lea     ecx, DWORD PTR _sum$[esp+12]
        push    ecx
        push    456                                     ; 000001c8H
        push    123                                     ; 0000007bH
        call    f1                      ; f1
        mov     edx, DWORD PTR _product$[esp+24]
        mov     eax, DWORD PTR _sum$[esp+24]
        push    edx
        push    eax
        push    OFFSET $SG3863
        call    _printf

...
```

1.7.1 C++ references

In C++, references are pointers as well, but they are called *safe*, because it's harder to make a mistake while working with them [ISO13, 8.3.2]. For example, reference should be always be pointing to the object of corresponding type and can't be NULL [Cli, 8.6]. Even more, reference cannot be changed, it's not possible to point to to another object (reseat) [Cli, 8.5].

If we will try to change are example to use references instead of pointers:

```
void f2 (int x, int y, int & sum, int & product)
{
        sum=x+y;
        product=x*y;
};
```

Then we'll figure out that compiled code is just the same:

Listing 1.16: Optimizing MSVC 2010

```
_x$ = 8                                                 ; size = 4
_y$ = 12                                                ; size = 4
_sum$ = 16                                              ; size = 4
_product$ = 20                                          ; size = 4
?f2@@YAXHHAAH0@Z PROC                                   ; f2
        mov     ecx, DWORD PTR _y$[esp-4]
        mov     eax, DWORD PTR _x$[esp-4]
        lea     edx, DWORD PTR [eax+ecx]
        imul eax, ecx
        mov ecx, DWORD PTR _product$[esp-4]
        push esi
        mov     esi, DWORD PTR _sum$[esp]
        mov     DWORD PTR [esi], edx
        mov     DWORD PTR [ecx], eax
        pop     esi
        ret     0
?f2@@YAXHHAAH0@Z ENDP                                   ; f2
```

(A reason why C++ functions has such strange names, will be described later 1.17.1.)

1.8 Conditional jumps

Now about conditional jumps.

```
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;
};
```

1.8.1 x86

x86 + MSVC

What we have in `f_signed()` function:

Listing 1.17: MSVC

```
_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
```

First instruction JLE mean *Jump if Larger or Equal*. In other words, if second operand is larger than first or equal, control flow will be passed to address or label mentioned in instruction. But if this condition will not trigger (second operand less than first), control flow will not be altered and first `printf()` will be called. The second check is JNE: *Jump if Not Equal*. Control flow will not altered if operands are equals to each other. The third check is JGE: *Jump if Greater or Equal* — jump if second operand is larger than first or they are equals to each other. By the way, if all three conditional jumps are triggered, no `printf()` will be called whatsoever. But, without special intervention, it is nearly impossible.

`f_unsigned()` function is the same, with the exception that JBE and JAE instructions are used here instead of JLE and JGE, see below about it:

GCC

GCC 4.4.1 produce almost the same code, but with `puts()` 1.1.2 instead of `printf()`.

Now let's take a look of `f_unsigned()` produced by GCC:

Listing 1.18: 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
```

Almost the same, with exception of instructions: JBE — *Jump if Below or Equal* and JAE — *Jump if Above or Equal*. These instructions (JA/JAE/JBE/JBE) are distinct from JG/JGE/JL/JLE in that way, they works with unsigned numbers.

See also section about signed number representations 2.4. So, where we see usage of JG/JL instead of JA/JBE or otherwise, we can almost be sure about signed or unsigned type of variable.

Here is also `main()` function, where nothing new to us:

Listing 1.19: `main()`

```
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
```

1.8.2 ARM

Optimizing Keil + ARM mode

Listing 1.20: Optimizing Keil + ARM mode

```
.text:000000B8                         EXPORT f_signed
.text:000000B8             f_signed                    ; CODE XREF: main+C
.text:000000B8 70 40 2D E9             STMFD   SP!, {R4-R6,LR}
.text:000000BC 01 40 A0 E1             MOV     R4, R1
.text:000000C0 04 00 50 E1             CMP     R0, R4
.text:000000C4 00 50 A0 E1             MOV     R5, R0
.text:000000C8 1A 0E 8F C2             ADRGT   R0, aAB         ; "a>b\n"
.text:000000CC A1 18 00 CB             BLGT    __2printf
.text:000000D0 04 00 55 E1             CMP     R5, R4
.text:000000D4 67 0F 8F 02             ADREQ   R0, aAB_0       ; "a==b\n"
.text:000000D8 9E 18 00 0B             BLEQ    __2printf
.text:000000DC 04 00 55 E1             CMP     R5, R4
.text:000000E0 70 80 BD A8             LDMGEFD SP!, {R4-R6,PC}
.text:000000E4 70 40 BD E8             LDMFD   SP!, {R4-R6,LR}
.text:000000E8 19 0E 8F E2             ADR     R0, aAB_1       ; "a<b\n"
.text:000000EC 99 18 00 EA             B       __2printf
.text:000000EC             ; End of function f_signed
```

A lot of instructions in ARM mode can be executed only when specific flags are set. This is often used while numbers comparing, for example.

For instance, ADD instruction is ADDAL internally in fact, where AL meaning *Always*, i.e., execute always. Predicates are encoded in 4 high bits of 32-bit ARM instructions (*condition field*). B instruction of unconditional jump is in fact conditional and encoded just like any other conditional jumps, but has AL in *condition field*, and what it means, executing always, ignoring flags.

ADRGT instructions works just like ADR, but will execute only in the case when previous CMP instruction, while comparing two numbers, found one number greater than another (*Greater Than*).

The next BLGT instruction behaves exactly as BL and will be triggered only if result of comparison was the same (*Greater Than*). ADRGT writes a pointer to the string ‘‘a>b\n’’, into R0 and BLGT calls `printf()`. Consequently, these instructions with -GT suffix, will be executed only in the case when value in R0 (a is there) was bigger than value in R4 (b is there).

Then we see ADREQ and BLEQ instructions. They behave just like ADR and BL, but will be executed only in the case when operands were equal to each other while comparison. Another CMP is before them (because `printf()` call may tamper state of flags).

Then we see LDMGEFD, this instruction works just like LDMFD[45], but will be triggered only in the case when one value was greater or equal to another while comparision (*Greater or Equal*).

The sense of ‘‘LDMGEFD SP!, {R4-R6,PC}’’ instruction is that is like function epilogue, but it will be triggered only if $a >= b$, only then function execution will be finished. But if it’s not true, i.e., $a < b$, then control flow come to next ‘‘LDMFD SP!, {R4-R6,LR}’’ instruction, this is one more function epilogue, this instruction restores R4-R6 registers state, but also LR instead of PC, thus, it doesn’t returns from function. Last two instructions calls `printf()` with the string «a<b\n» as sole argument. Unconditional jump to `printf()` instead of function return, is what we already examined in «`printf()` with several arguments» section, here 1.3.2.

[45] Load Multiple Full Descending

`f_unsigned` is similar, but `ADRHI`, `BLHI`, and `LDMCSFD` instructions are used there, these predicates (*HI = Unsigned higher, CS = Carry Set (greater than or equal)*) are analogical to those examined before, but serving for unsigned values.

There are not much new in `main()` for us:

Listing 1.21: `main()`

```
.text:00000128                             EXPORT main
.text:00000128             main
.text:00000128 10 40 2D E9                 STMFD   SP!, {R4,LR}
.text:0000012C 02 10 A0 E3                 MOV     R1, #2
.text:00000130 01 00 A0 E3                 MOV     R0, #1
.text:00000134 DF FF FF EB                 BL      f_signed
.text:00000138 02 10 A0 E3                 MOV     R1, #2
.text:0000013C 01 00 A0 E3                 MOV     R0, #1
.text:00000140 EA FF FF EB                 BL      f_unsigned
.text:00000144 00 00 A0 E3                 MOV     R0, #0
.text:00000148 10 80 BD E8                 LDMFD   SP!, {R4,PC}
.text:00000148             ; End of function main
```

That's how to get rid of conditional jumps in ARM mode.

Why it's so good? Because, ARM is RISC-processor with pipeline for instructions executing. In short, pipelined processor is not very good on jumps at all, so that's why branch predictors are critical here. It's very good if the program has as few jumps as possible, conditional and unconditional, so that's why, predicated instructions can help in reducing conditional jumps count.

There are no such feature in x86, if not to count `CMOVcc` instruction, it's the same as `MOV`, but it's triggered only when specific flags are set, usually, set while comparing by `CMP`.

Optimizing Keil + thumb mode

Listing 1.22: Optimizing Keil + thumb mode

```
.text:00000072             f_signed                        ; CODE XREF: main+6
.text:00000072 70 B5                       PUSH    {R4-R6,LR}
.text:00000074 0C 00                       MOVS    R4, R1
.text:00000076 05 00                       MOVS    R5, R0
.text:00000078 A0 42                       CMP     R0, R4
.text:0000007A 02 DD                       BLE     loc_82
.text:0000007C A4 A0                       ADR     R0, aAB         ; "a>b\n"
.text:0000007E 06 F0 B7 F8                 BL      __2printf
.text:00000082
.text:00000082             loc_82                                  ; CODE XREF: f_signed+8
.text:00000082 A5 42                       CMP     R5, R4
.text:00000084 02 D1                       BNE     loc_8C
.text:00000086 A4 A0                       ADR     R0, aAB_0       ; "a==b\n"
.text:00000088 06 F0 B2 F8                 BL      __2printf
.text:0000008C
.text:0000008C             loc_8C                                  ; CODE XREF: f_signed+12
.text:0000008C A5 42                       CMP     R5, R4
.text:0000008E 02 DA                       BGE     locret_96
.text:00000090 A3 A0                       ADR     R0, aAB_1       ; "a<b\n"
.text:00000092 06 F0 AD F8                 BL      __2printf
.text:00000096
.text:00000096             locret_96                               ; CODE XREF: f_signed+1C
.text:00000096 70 BD                       POP     {R4-R6,PC}
.text:00000096             ; End of function f_signed
```

Only `B` instructions in thumb mode may be supplemented by *condition codes*, so the thumb code looks more ordinary.

`BLE` is usual conditional jump *Less than or Equal*, `BNE` — *Not Equal*, `BGE` — *Greater than or Equal*.

`f_unsigned` function is just the same, but other instructions are used while working with unsigned values:`BLS` (*Unsigned lower or same*) and `BCS` (*Carry Set (Greater than ir equal)*).

1.9 switch()/case/default

1.9.1 Few number of cases

```
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;
    };
};
```

x86

Result (MSVC 2010):

Listing 1.23: 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
```

Out function with a few cases in switch(), in fact, is analogous to this construction:

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

```
    if (a==0)
        printf ("zero\n");
    else if (a==1)
        printf ("one\n");
    else if (a==2)
        printf ("two\n");
    else
        printf ("something unknown\n");
};
```

When few cases in switch(), and we see such code, it's impossible to say with certainty, was it switch() in source code, or just pack of if(). This mean, switch() is syntactic sugar for large number of nested checks constructed using if().

Nothing specially new to us in generated code, with the exception that compiler moving input variable a to temporary local variable tv64.

If to compile the same in GCC 4.4.1, we'll get alsmost the same, even with maximal optimization turned on (-O3 option).

Now let's turn on optimization in MSVC (/Ox): `cl 1.c /Fa1.asm /Ox`

Listing 1.24: MSVC

```
_a$ = 8                             ; size = 4
_f    PROC
    mov    eax, DWORD PTR _a$[esp-4]
    sub    eax, 0
    je     SHORT $LN4@f
    sub    eax, 1
    je     SHORT $LN3@f
    sub    eax, 1
    je     SHORT $LN2@f
    mov    DWORD PTR _a$[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
```

Here we can see even dirty hacks.

First: a is placed into EAX and 0 subtracted from it. Sounds absurdly, but it may need to check if 0 was in EAX before? If yes, flag ZF will be set (this also mean that subtracting from zero is zero) and first conditional jump JE (*Jump if Equal* or synonym JZ — *Jump if Zero*) will be triggered and control flow passed to $LN4@f label, where 'zero' message is begin printed. If first jump was not triggered, 1 subtracted from input value and if at some stage 0 will be resulted, corresponding jump will be triggered.

And if no jump triggered at all, control flow passed to printf() with argument 'something unknown'.

Second: we see unusual thing for us: string pointer is placed into a variable, and then printf() is called not via CALL, but via JMP. This could be explained simply. Caller pushing to stack some value and via CALL calling our function. CALL itself pushing returning address to stack and do unconditional jump to our function address. Our function at any place of its execution (since it do not contain any instruction moving stack pointer) has the following stack layout:

- ESP — pointing to return address
- ESP+4 — pointing to a variable

On the other side, when we need to call printf() here, we need exactly the same stack layout, except of first printf() argument pointing to string. And that is what our code does.

It replaces function's first argument to different and jumping to `printf()`, as if not our function `f()` was called firstly, but immediately `printf()`. `printf()` printing some string to `stdout` and then execute `RET` instruction, which POPping return address from stack and control flow is returned not to `f()`, but to `f()`'s callee, escaping `f()`.

All it's possible because `printf()` is called right at the end of `f()` in any case. In some way, it's all similar to `longjmp()`[46]. And of course, it's all done for the sake of speed.

Similar case with ARM compiler described in "`printf()` with several arguments", section, here 1.3.2.

ARM: Optimizing Keil + ARM mode

```
.text:0000014C             f1
.text:0000014C 00 00 50 E3           CMP     R0, #0
.text:00000150 13 0E 8F 02           ADREQ   R0, aZero       ; "zero\n"
.text:00000154 05 00 00 0A           BEQ     loc_170
.text:00000158 01 00 50 E3           CMP     R0, #1
.text:0000015C 4B 0F 8F 02           ADREQ   R0, aOne        ; "one\n"
.text:00000160 02 00 00 0A           BEQ     loc_170
.text:00000164 02 00 50 E3           CMP     R0, #2
.text:00000168 4A 0F 8F 12           ADRNE   R0, aSomethingUnkno ; "something unknown\n"
.text:0000016C 4E 0F 8F 02           ADREQ   R0, aTwo        ; "two\n"
.text:00000170
.text:00000170             loc_170                         ; CODE XREF: f1+8
.text:00000170                                             ; f1+14
.text:00000170 78 18 00 EA           B       __2printf
```

Again, by investigating this code, we cannot say, was it switch() in the original source code, or pack of if() statements.

Anyway, we see here predicated instructions again, like `ADREQ` (*Equal*), which will be triggered only in $R0 = 0$ case, and the, address of *«zero\n»* string will be loaded into `R0`. The next instruction `BEQ` (*(Branch Equal)*) will redirect control flow to `loc_170`, if $R0 = 0$. By the way, observant reader may ask, will `BEQ` triggered right, because `ADREQ` before it is already filled `R0` register with some other value. Yes, it will, because `BEQ` checking flags set by `CMP` instruction, and `ADREQ` not modifying flags at all.

By the way, there are *-S* suffix for some instructions in ARM, indicative that instruction will not modify flags. For example `ADDS` Will add two number, but flags will not be touched. Such instructions are convenient to use between `CMP` where flags are set and, for example, conditional jumps, where flags are used.

Other instructions are already familiar to us. There are only one call to `printf()`, at the end, and we already examined this trick here 1.3.2. There are three paths to `printf()` at the end.

Also pay attention to what is going on if $a = 2$ and if a is not in range of constants it's comparing against. ''`CMP R0, #2`'' instruction is needed here to know, if $a = 2$ or not. If it's not true, then `ADRNE` will load pointer to the string *«something unknown \n»* into `R0`, because, a was already checked before to be equal to 0 or 1, so we can be assured that a is not equal to these numbers at this point. And if $R0 = 2$, a pointer to string *«two\n»* will be loaded by `ADREQ` into `R0`.

ARM: Optimizing Keil + thumb mode

```
.text:000000D4             f1
.text:000000D4 10 B5                 PUSH    {R4,LR}
.text:000000D6 00 28                 CMP     R0, #0
.text:000000D8 05 D0                 BEQ     zero_case
.text:000000DA 01 28                 CMP     R0, #1
.text:000000DC 05 D0                 BEQ     one_case
.text:000000DE 02 28                 CMP     R0, #2
.text:000000E0 05 D0                 BEQ     two_case
.text:000000E2 91 A0                 ADR     R0, aSomethingUnkno ; "something unknown\n"
.text:000000E4 04 E0                 B       default_case
.text:000000E6             ; ---------------------------------------------------------------------------
.text:000000E6             zero_case                           ; CODE XREF: f1+4
```

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

```
.text:000000E6 95 A0                    ADR     R0, aZero       ; "zero\n"
.text:000000E8 02 E0                    B       default_case
.text:000000EA              ; ---------------------------------------------------------------------------
.text:000000EA              one_case                                ; CODE XREF: f1+8
.text:000000EA 96 A0                    ADR     R0, aOne        ; "one\n"
.text:000000EC 00 E0                    B       default_case
.text:000000EE              ; ---------------------------------------------------------------------------
.text:000000EE              two_case                                ; CODE XREF: f1+C
.text:000000EE 97 A0                    ADR     R0, aTwo        ; "two\n"
.text:000000F0              default_case                            ; CODE XREF: f1+10
.text:000000F0                                                      ; f1+14
.text:000000F0 06 F0 7E F8              BL      __2printf
.text:000000F4 10 BD                    POP     {R4,PC}
.text:000000F4              ; End of function f1
```

As I already mentioned, there are no feature of *connecting* predicates to majority of functions in thumb mode, so the thumb-code here is somewhat like the x86 code, easily understandable.

1.9.2 A lot of cases

If `switch()` statement contain a lot of case's, it is not very convenient for compiler to emit too large code with a lot JE/JNE instructions.

```
void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    case 3: printf ("three\n"); break;
    case 4: printf ("four\n"); break;
    default: printf ("something unknown\n"); break;
    };
};
```

x86

We got (MSVC 2010):

Listing 1.25: 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
```

OK, what we see here is: there are a set of `printf()` calls with various arguments. All them has not only addresses in process memory, but also internal symbolic labels assigned by compiler. Besides, all these labels are also places into `$LN11@f` internal table.

At the function beginning, if `a` is greater than 4, control flow is passed to label `$LN1@f`, where `printf()` with argument `'something unknown'` is called.

And if `a` value is less or equals to 4, let's multiply it by 4 and add `$LN1@f` table address. That is how address inside of table is constructed, pointing exactly to the element we need. For example, let's say `a` is equal to 2. $2*4=8$ (all table elements are addresses within 32-bit process, that is why all elements contain 4 bytes). Address of `$LN11@f` table + 8 = it will be table element where `$LN4@f` label is stored. `JMP` fetch `$LN4@f` address from the table and jump to it.

This table called sometimes *jumptable*.

Then corresponding `printf()` is called with argument `'two'`. Literally, `jmp DWORD PTR $LN11@f[ecx*4]` instruction mean *jump to DWORD, which is stored at address* `$LN11@f + ecx * 4`.

`npad` 2.3 is assembly language macro, aligning next label so that it will be stored at address aligned by 4 bytes (or 16). This is very suitable for processor, because it can fetch 32-bit values from memory through memory bus, cache memory, etc, in much effective way if it is aligned.

Let's see what GCC 4.4.1 generate:

Listing 1.26: 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
```

It is almost the same, except little nuance: argument `arg_0` is multiplied by 4 with by shifting it to left by 2 bits (it is almost the same as multiplication by 4) 1.15.3. Then label address is taken from `off_804855C` array, address calculated and stored into `EAX`, then ‘‘`JMP EAX`’’ do actual jump.

ARM: Optimizing Keil + ARM mode

```
00000174            f2
00000174 05 00 50 E3            CMP     R0, #5          ; switch 5 cases
00000178 00 F1 8F 30            ADDCC   PC, PC, R0,LSL#2 ; switch jump
0000017C 0E 00 00 EA            B       default_case    ; jumptable 00000178 default case
00000180            ; ---------------------------------------------------------------------------
00000180
00000180            loc_180                             ; CODE XREF: f2+4
00000180 03 00 00 EA            B       zero_case       ; jumptable 00000178 case 0
00000184            ; ---------------------------------------------------------------------------
00000184
00000184            loc_184                             ; CODE XREF: f2+4
00000184 04 00 00 EA            B       one_case        ; jumptable 00000178 case 1
00000188            ; ---------------------------------------------------------------------------
```

```
00000188
00000188             loc_188                                 ; CODE XREF: f2+4
00000188 05 00 00 EA                 B       two_case        ; jumptable 00000178 case 2
0000018C             ; ---------------------------------------------------------------------------
0000018C
0000018C             loc_18C                                 ; CODE XREF: f2+4
0000018C 06 00 00 EA                 B       three_case      ; jumptable 00000178 case 3
00000190             ; ---------------------------------------------------------------------------
00000190
00000190             loc_190                                 ; CODE XREF: f2+4
00000190 07 00 00 EA                 B       four_case       ; jumptable 00000178 case 4
00000194             ; ---------------------------------------------------------------------------
00000194
00000194             zero_case                               ; CODE XREF: f2+4
00000194                                                     ; f2:loc_180
00000194 EC 00 8F E2                 ADR     R0, aZero       ; jumptable 00000178 case 0
00000198 06 00 00 EA                 B       loc_1B8
0000019C             ; ---------------------------------------------------------------------------
0000019C
0000019C             one_case                                ; CODE XREF: f2+4
0000019C                                                     ; f2:loc_184
0000019C EC 00 8F E2                 ADR     R0, aOne        ; jumptable 00000178 case 1
000001A0 04 00 00 EA                 B       loc_1B8
000001A4             ; ---------------------------------------------------------------------------
000001A4
000001A4             two_case                                ; CODE XREF: f2+4
000001A4                                                     ; f2:loc_188
000001A4 01 0C 8F E2                 ADR     R0, aTwo        ; jumptable 00000178 case 2
000001A8 02 00 00 EA                 B       loc_1B8
000001AC             ; ---------------------------------------------------------------------------
000001AC
000001AC             three_case                              ; CODE XREF: f2+4
000001AC                                                     ; f2:loc_18C
000001AC 01 0C 8F E2                 ADR     R0, aThree      ; jumptable 00000178 case 3
000001B0 00 00 00 EA                 B       loc_1B8
000001B4             ; ---------------------------------------------------------------------------
000001B4
000001B4             four_case                               ; CODE XREF: f2+4
000001B4                                                     ; f2:loc_190
000001B4 01 0C 8F E2                 ADR     R0, aFour       ; jumptable 00000178 case 4
000001B8
000001B8             loc_1B8                                 ; CODE XREF: f2+24
000001B8                                                     ; f2+2C
000001B8 66 18 00 EA                 B       __2printf
000001BC             ; ---------------------------------------------------------------------------
000001BC
000001BC             default_case                            ; CODE XREF: f2+4
000001BC                                                     ; f2+8
000001BC D4 00 8F E2                 ADR     R0, aSomethingUnkno ; jumptable 00000178 default case
000001C0 FC FF FF EA                 B       loc_1B8
000001C0             ; End of function f2
```

This code makes use that ARM mode feature, all instructions in this mode has size 4 bytes.

Let's keep in mind that maximum value for a is 4 and any greater value should cause *«something unknown\n»* string printing.

The very first ''`CMP R0, #5`'' instruction compares a input value with 5.

The next ''`ADDCC PC, PC, R0,LSL#2`'' [47] instruction will execute only if $R0 < 5$ (*CC=Carry clear / Less than*). Consequently, if `ADDCC` will not trigger (it's a $R0 \geq 5$ case), a jump to *default_case*label will be occured.

But if $R0 < 5$ and `ADDCC` will trigger, following events will happen:

Value in `R0` is multiplied by 4. In fact, `LSL#2` at the instruction's ending mean "shift left by 2 bits". But as we will see later 1.15.3 in "Shifts" section, shift left by 2 bits is just equivalently to multiplying by 4.

[47] ADD — addition

Then, $R0 * 4$ value we got, is added to current PC value, thus jumping to one of B (*Branch*) instructions located below.

At the moment of ADDCC execution, PC value is 8 bytes ahead ($0x180$) than address at which ADDCC instruction is located ($0x178$), or, in other words, 2 instructions ahead.

This is how ARM processor pipeline works: when ADDCC instruction is executed, the processor at that moment is beginning to process instruction after the next one, so that's why PC pointing there.

If $a = 0$, then nothing will be added to PC, and actual PC value will be written into PC (which is 8 bytes ahead) and jump to label *loc_180* will be happen, this is 8 bytes ahead of the place where ADDCC instruction is.

In case of $a = 1$, then $PC + 8 + a * 4 = PC + 8 + 1 * 4 = PC + 16 = 0x184$will be written to PC, this is address of *loc_184* label.

With every 1 added to a, resulting PC increasing by 4. 4 is also instruction length in ARM mode and also, length of each B instruction length, there are 5 of them in row.

Each of these five B instructions passing control further, where something is going on, what was programmed in *switch()*. Pointer loading to corresponding string occuring there, etc.

ARM: Optimizing Keil + thumb mode

```
000000F6                                EXPORT f2
000000F6                    f2
000000F6 10 B5                          PUSH    {R4,LR}
000000F8 03 00                          MOVS    R3, R0
000000FA 06 F0 69 F8                    BL      __ARM_common_switch8_thumb ; switch 6 cases
000000FA                    ; ---------------------------------------------------------------------------
000000FE 05                             DCB 5
000000FF 04 06 08 0A 0C 10              DCB 4, 6, 8, 0xA, 0xC, 0x10 ; jump table for switch statement
00000105 00                             ALIGN 2
00000106
00000106                    zero_case                               ; CODE XREF: f2+4
00000106 8D A0                          ADR     R0, aZero       ; jumptable 000000FA case 0
00000108 06 E0                          B       loc_118
0000010A                    ; ---------------------------------------------------------------------------
0000010A
0000010A                    one_case                                ; CODE XREF: f2+4
0000010A 8E A0                          ADR     R0, aOne        ; jumptable 000000FA case 1
0000010C 04 E0                          B       loc_118
0000010E                    ; ---------------------------------------------------------------------------
0000010E
0000010E                    two_case                                ; CODE XREF: f2+4
0000010E 8F A0                          ADR     R0, aTwo        ; jumptable 000000FA case 2
00000110 02 E0                          B       loc_118
00000112                    ; ---------------------------------------------------------------------------
00000112
00000112                    three_case                              ; CODE XREF: f2+4
00000112 90 A0                          ADR     R0, aThree      ; jumptable 000000FA case 3
00000114 00 E0                          B       loc_118
00000116                    ; ---------------------------------------------------------------------------
00000116
00000116                    four_case                               ; CODE XREF: f2+4
00000116 91 A0                          ADR     R0, aFour       ; jumptable 000000FA case 4
00000118
00000118                    loc_118                                 ; CODE XREF: f2+12
00000118                                                            ; f2+16
00000118 06 F0 6A F8                    BL      __2printf
0000011C 10 BD                          POP     {R4,PC}
0000011E                    ; ---------------------------------------------------------------------------
0000011E
0000011E                    default_case                            ; CODE XREF: f2+4
0000011E 82 A0                          ADR     R0, aSomethingUnkno ; jumptable 000000FA default case
00000120 FA E7                          B       loc_118


000061D0                                EXPORT __ARM_common_switch8_thumb
```

```
000061D0                    __ARM_common_switch8_thumb              ; CODE XREF: example6_f2+4
000061D0 78 47                          BX      PC
000061D0                    ; ---------------------------------------------------------------------------
000061D2 00 00                          ALIGN 4
000061D2                    ; End of function __ARM_common_switch8_thumb
000061D2
000061D4                                CODE32
000061D4
000061D4                    ; =============== S U B R O U T I N E =======================================
000061D4
000061D4
000061D4                    __32__ARM_common_switch8_thumb          ; CODE XREF: __ARM_common_switch8_thumb
000061D4 01 C0 5E E5                    LDRB    R12, [LR,#-1]
000061D8 0C 00 53 E1                    CMP     R3, R12
000061DC 0C 30 DE 27                    LDRCSB  R3, [LR,R12]
000061E0 03 30 DE 37                    LDRCCB  R3, [LR,R3]
000061E4 83 C0 8E E0                    ADD     R12, LR, R3,LSL#1
000061E8 1C FF 2F E1                    BX      R12
000061E8                    ; End of function __32__ARM_common_switch8_thumb
```

One cannot be sure all instructions in thumb and thumb-2 modes will have same size. It's even can be said that in these modes instructions has variable length, just like in x86.

So there are a special table added, containing information about how much cases are there, not including default-case, and offset, for each, each encoding a label, to which control should be passed in corresponding case.

A special function here present in order to work with the table and pass control, named *__ARM_common_switch8_thumb*. It is beginning with ‘‘`BX PC`’’ instruction, which function is to switch processor to ARM-mode. Then you may see the function for table processing. It's too complex for describing it here now, so I'll omit elaborations.

But it's interesting to note that the function uses `LR` reigster as a pointer to the table. Indeed, after this function calling, `LR` will contain address after
‘‘`BL __ARM_common_switch8_thumb`’’ instruction, and the table is beginning right there.

It's also worth noting that the code is generated as a separate function in order to reuse it, in similar places, in similar cases, for *switch()* processing, so compiler will not generate same code at each place.

IDA 5 successfully perceived it as a service function and table, automatically, and added commentaries to labels like `jumptable 000000FA case 0`.

1.10 Loops

1.10.1 x86

There is a special LOOP instruction in x86 instruction set, it is checking ECX register value and if it is not zero, do ECX decrement[48] and pass control flow to label mentioned in LOOP operand. Probably, this instruction is not very convenient, so, I didn't ever see any modern compiler emitting it automatically. So, if you see the instruction somewhere in code, it is most likely this is manually written piece of assembly code.

By the way, as home exercise, you could try to explain, why this instruction is not very convenient.

In C/C++ loops are constructed using `for()`, `while()`, `do/while()` statements.

Let's start with `for()`.

This statement defines loop initialization (set loop counter to initial value), loop condition (is counter is bigger than a limit?), what is done at each iteration (increment/decrement) and of course loop body.

```
for (initialization; condition; at each iteration)
{
  loop_body;
}
```

So, generated code will be consisted of four parts too.

Let's start with simple example:

```
int main()
{
        int i;

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

        return 0;
};
```

Result (MSVC 2010):

Listing 1.27: MSVC 2010

```
_i$ = -4
_main    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    DWORD PTR _i$[ebp], 2   ; loop initialization
    jmp    SHORT $LN3@main
$LN2@main:
    mov    eax, DWORD PTR _i$[ebp] ; here is what we do after each iteration:
    add    eax, 1                  ; add 1 to i value
    mov    DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp    DWORD PTR _i$[ebp], 10  ; this condition is checked *before* each iteration
    jge    SHORT $LN1@main         ; if i is biggest or equals to 10, let's finish loop
    mov    ecx, DWORD PTR _i$[ebp] ; loop body: call f(i)
    push   ecx
    call   _f
    add    esp, 4
    jmp    SHORT $LN2@main         ; jump to loop begin
$LN1@main:                         ; loop end
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
```

Nothing very special, as we see.

GCC 4.4.1 emitting almost the same code, with small difference:

[48] subtracting 1 from it

Listing 1.28: GCC 4.4.1

```
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
```

Now let's see what we will get if optimization is turned on (`/Ox`):

Listing 1.29: Optimizing MSVC

```
_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
```

What is going on here is: place for *i* variable is not allocated in local stack anymore, but even individual register: `ESI`. This is possible in such small functions where not so many local variables are present.

One very important property is that `f()` function should not change `ESI` value. Our compiler is sure here. And if compiler decided to use `ESI` in `f()` too, it would be saved then at `f()` function prologue and restored at `f()` epilogue. Almost like in our listing: please note `PUSH ESI/POP ESI` at the function begin and end.

Let's try GCC 4.4.1 with maximal optimization turned on (`-O3` option):

Listing 1.30: Optimizing GCC 4.4.1

```
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
```

Huh, GCC just unwind our loop.

Loop unwinding has advantage in these cases when there are not so much iterations and we could economy some execution speed by removing all loop supporting instructions. On the other side, resulting code is obviously larger.

OK, let's increase maximal value of *i* to 100 and try again. GCC resulting:

Listing 1.31: GCC

```
                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
```

It's quite similar to what MSVC 2010 with optimization (`/Ox`) produce. With the exception that `EBX` register will be fixed to `i` variable. GCC is sure this register will not be modified inside of `f()` function, and if it will, it will be saved at the function prologue and restored at epilogue, just like here in `main()`.

1.10.2 ARM

Non-optimizing Keil + ARM mode

```
main
                STMFD   SP!, {R4,LR}
                MOV     R4, #2
```

```
                B       loc_368
; ---------------------------------------------------------------------------

loc_35C                                 ; CODE XREF: main+1C
                MOV     R0, R4
                BL      f
                ADD     R4, R4, #1

loc_368                                 ; CODE XREF: main+8
                CMP     R4, #0xA
                BLT     loc_35C
                MOV     R0, #0
                LDMFD   SP!, {R4,PC}
```

Iteration counter `i` will be stored in `R4` register.

''`MOV R4, #2`'' instruction just initializing *i*.

''`MOV R0, R4`'' and ''`BL f`'' instructions are compose loop body, the first instruction preparing argument for `f()` function and the second is calling it.

''`ADD R4, R4, #1`'' instruction is just adding 1 to *i* during each iteration.

''`CMP R4, #0xA`'' comparing `i` with $0xA$ (10). Next instruction `BLT` (*Branch Less Than*) will jump if *i* is less than 10.

Otherwise, 0 will be written into `R0` (because our function returning 0) and function execution ended.

Optimizing Keil + thumb mode

```
_main
                PUSH    {R4,LR}
                MOVS    R4, #2

loc_132                                 ; CODE XREF: _main+E
                MOVS    R0, R4
                BL      example7_f
                ADDS    R4, R4, #1
                CMP     R4, #0xA
                BLT     loc_132
                MOVS    R0, #0
                POP     {R4,PC}
```

Practically, the same.

Optimizing Xcode (LLVM) + thumb-2 mode

```
_main
        PUSH            {R4,R7,LR}
        MOVW            R4, #0x1124 ; "%d\n"
        MOVS            R1, #2
        MOVT.W          R4, #0
        ADD             R7, SP, #4
        ADD             R4, PC
        MOV             R0, R4
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #3
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #4
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #5
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #6
```

```
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #7
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #8
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #9
        BLX             _printf
        MOVS            R0, #0
        POP             {R4,R7,PC}
```

In fact, this was in my `f()` function:

```
void f(int i)
{
    // do something here
    printf ("%d\n", i);
};
```

So, LLVM not just *unrolled* the loop, but also represented my very simple function `f()` as *inlined*, and inserted its body 8 times instead of loop. This is possible when function is so primitive, like mine, and when it's called not many times, like here.

1.10.3 One more thing

On the code generated we can see: after *i*initialization, loop body will not be executed, but *i* condition checked first, and only after loop body is to be executed. And that's correct. Because, if loop condition is not met at the beginning, loop body shouldn't be executed. For example, this is possible in the following case:

```
for (i; i<total_entries_to_process; i++)
    loop_body;
```

If *total_entries_to_process* equals to zero, loop body shouldn't be executed whatsoever. So that's why condition checked before loop body execution.

However, optimizing compiler may swap condition check and loop body, if it sure that the situation described here is not possible, like in case of our very simple example and Keil, Xcode (LLVM), MSVC in optimization mode.

1.11 strlen()

Now let's talk about loops one more time. Often, `strlen()` function[49] is implemented using `while()` statement. Here is how it's done in MSVC standard libraries:

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

        while( *eos++ ) ;

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

1.11.1 x86

Let's compile:

```
_eos$ = -4                      ; size = 4
_str$ = 8                       ; size = 4
_strlen PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _str$[ebp]  ; place pointer to string from str
    mov     DWORD PTR _eos$[ebp], eax  ; place it to local varuable eos
$LN2@strlen_:
    mov     ecx, DWORD PTR _eos$[ebp]  ; ECX=eos

    ; take 8-bit byte from address in ECX and place it as 32-bit value to EDX with sign extension

    movsx   edx, BYTE PTR [ecx]
    mov     eax, DWORD PTR _eos$[ebp]  ; EAX=eos
    add     eax, 1                     ; increment EAX
    mov     DWORD PTR _eos$[ebp], eax  ; place EAX back to eos
    test    edx, edx                   ; EDX is zero?
    je      SHORT $LN1@strlen_         ; yes, then finish loop
    jmp     SHORT $LN2@strlen_         ; continue loop
$LN1@strlen_:

    ; here we calculate the difference between two pointers

    mov    eax, DWORD PTR _eos$[ebp]
    sub    eax, DWORD PTR _str$[ebp]
    sub    eax, 1                      ; subtract 1 and return result
    mov    esp, ebp
    pop    ebp
    ret    0
_strlen_ ENDP
```

Two new instructions here: `MOVSX` 1.11.1 and `TEST`.

About first: `MOVSX` 1.11.1 is intended to take byte from some place and store value in 32-bit register. `MOVSX` 1.11.1 meaning *MOV with Sign-Extent*. Rest bits starting at 8th till 31th `MOVSX` 1.11.1 will set to 1 if source byte in memory has *minus* sign or to 0 if *plus*.

And here is why all this.

C/C++ standard defines *char* type as signed. If we have two values, one is *char* and another is *int*, (*int* is signed too), and if first value contain -2 (it is coded as $0xFE$) and we just copying this byte into *int* container, there will be $0x000000FE$, and this, from the point of signed *int* view is 254, but not -2. In signed int, -2 is coded as $0xFFFFFFFE$. So if we need to transfer $0xFE$ value from variable of *char* type to *int*, we need to identify its sign and extend it. That is what `MOVSX` 1.11.1 does.

See also in section "*Signed number representations*" 2.4.

[49] counting characters in string in C language

I'm not sure if compiler need to store *char* variable in EDX, it could take 8-bit register part (let's say DL). Apparently, compiler's register allocator[50] works like that.

Then we see TEST EDX, EDX. About TEST instruction, read more in section about bit fields 1.15. But here, this instruction just checking EDX value, if it is equals to 0.

Let's try 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
```

The result almost the same as MSVC did, but here we see MOVZX isntead of MOVSX 1.11.1. MOVZX mean *MOV with Zero-Extent*. This instruction place 8-bit or 16-bit value into 32-bit register and set the rest bits to zero. In fact, this instruction is convenient only because it enable us to replace two instructions at once: xor eax, eax / mov al, [...].

On other hand, it is obvious to us that compiler could produce that code: mov al, byte ptr [eax] / test al, al — it is almost the same, however, highest EAX register bits will contain random noise. But let's think it is compiler's drawback — it can't produce more understandable code. Strictly speaking, compiler is not obliged to emit understandable (to humans) code at all.

Next new instruction for us is SETNZ. Here, if AL contain not zero, test al, al will set zero to ZF flag, but SETNZ, if ZF==0 (*NZ* mean *not zero*) will set 1 to AL. Speaking in natural language, *if AL is not zero, let's jump to loc_80483F0*. Compiler emitted slightly redundant code, but let's not forget that optimization is turned off.

Now let's compile all this in MSVC 2010, with optimization turned on (/Ox):

```
_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
```

[50] compiler's function assigning local variables to CPU registers

Now it's all simpler. But it is needless to say that compiler could use registers such efficiently only in small functions with small number of local variables.

INC/DEC — are increment/decrement instruction, in other words: add 1 to variable or subtract.

Let's check GCC 4.4.1 with optimization turned on (-O3 key):

```
                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
```

Here GCC is almost the same as MSVC, except of MOVZX presence.

However, MOVZX could be replaced here to `mov dl, byte ptr [eax]`.

Probably, it is simpler for GCC compiler's code generator to *remember* that whole register is allocated for *char* variable and it can be sure that highest bits will not contain noise at any point.

After, we also see new instruction NOT. This instruction inverts all bits in operand. It can be said, it is synonym to `XOR ECX, 0ffffffffh` instruction. NOT and following ADD calculating pointer difference and subtracting 1. At the beginning ECX, where pointer to str is stored, inverted and 1 is subtracted from it.

See also: "Signed number representations" 2.4.

In other words, at the end of function, just after loop body, these operations are executed:

```
ecx=str;
eax=eos;
ecx=(-ecx)-1;
eax=eax+ecx
return eax
```

...and this is equivalent to:

```
ecx=str;
eax=eos;
eax=eax-ecx;
eax=eax-1;
return eax
```

Why GCC decided it would be better? I cannot be sure. But I'm assure that both variants are equivalent in efficiency sense.

1.11.2 ARM

Non-optimizing Xcode (LLVM) + ARM mode

Listing 1.32: Non-optimizing Xcode (LLVM) + ARM mode

```
_strlen

eos         = -8
str         = -4
```

```
            SUB     SP, SP, #8 ; allocate 8 bytes for local variables
            STR     R0, [SP,#8+str]
            LDR     R0, [SP,#8+str]
            STR     R0, [SP,#8+eos]

loc_2CB8                    ; CODE XREF: _strlen+28
            LDR     R0, [SP,#8+eos]
            ADD     R1, R0, #1
            STR     R1, [SP,#8+eos]
            LDRSB   R0, [R0]
            CMP     R0, #0
            BEQ     loc_2CD4
            B       loc_2CB8
; ---------------------------------------------------------------

loc_2CD4                    ; CODE XREF: _strlen+24
            LDR     R0, [SP,#8+eos]
            LDR     R1, [SP,#8+str]
            SUB     R0, R0, R1 ; R0=eos-str
            SUB     R0, R0, #1 ; R0=R0-1
            ADD     SP, SP, #8 ; deallocate 8 bytes for local variables
            BX      LR
```

Non-optimizing LLVM generates too much code, however, here we can see how function works with local variables in stack. There are only two local variables in our function, *eos* and *str*.

In this listing, generated by IDA 5, I renamed *var_8* and *var_4* into *eos* and *str* manually.

So, first instructions are just saves input value in *str* and *eos*.

Loop body is beginning at *loc_2CB8* label.

First three instruction in loop body (`LDR`, `ADD`, `STR`) loads *eos* value into `R0`, then value is incremented and it's saving back into *eos* local variable located in stack.

The next "`LDRSB R0, [R0]`" (*Load Register Signed Byte*) instruction loading byte from memory at `R0` address and sign-extends it to 32-bit. This is similar to `MOVSX` 1.11.1 instruction in x86. The compiler treating this byte as signed because *char* type in C standard is signed. I already wrote about it 1.11.1 in this section, but related to x86.

It's should be noted, there are to way to use 8-bit part or 16-bit part of 32-bit register in ARM, as it's possible in x86. Apparently, it's because x86 has a huge history of compatibility with its ancestors like 16-bit 8086 and even 8-bit 8080, but ARM was developed from scratch as 32-bit RISC-processor. Consequently, in order to process separate bytes in ARM, one have to use 32-bit registers anyway.

So, `LDRSB` loads symbol from string into `R0`, one by one. Next `CMP` and `BEQ` instructions checks, if loaded symbol is zero. If not zero, control passing to loop body begin. And if zero, loop is finishing.

At the end of function, a difference between *eos* and *str* is calculated, 1 is also subtracting, and resulting value is returned via `R0`.

By the way, please note, registers wasn't saved in this function. That's because by ARM calling convention, `R0`-`R3` registers are "scratch registers", they are intended for arguments passing, its values may not be restored upon function exit, because calling function will not use them anymore. Consequently, they may be used for anything we want. Other registers are not used here, so that's why we have nothing to save in stack. Thus, control may be returned back to calling function by simple jump (`BX`), to address in `LR` register.

Optimizing Xcode (LLVM) + thumb mode

Listing 1.33: Optimizing Xcode (LLVM) + thumb mode

```
_strlen
            MOV     R1, R0

loc_2DF6                    ; CODE XREF: _strlen+8
            LDRB.W  R2, [R1],#1
            CMP     R2, #0
            BNE     loc_2DF6
            MVNS    R0, R0
            ADD     R0, R1
```

```
        BX      LR
```

As optimizing LLVM concludes, place in stack for *eos* and *str* may not be allocated, and these variables may always be stored right in registers. Before loop body beginning, *str* will always be in R0, and *eos* — in R1.

''LDRB.W R2, [R1],#1'' instruction loads byte from memory at the address R1 into R2, sign-extending it to 32-bit value, but not only that. #1 at the instruction's end calling "Post-indexed addressing", this mean, 1 is to be added to R1 after byte load. That's convenient when accessing arrays.

There are no such addressing mode in x86, but it's present in some other processors, even on PDP-11. There is a legend that pre-increment, post-increment, pre-decrement and post-decrement modes in PDP-11, were "guilty" in appearance such C language (which developed on PDP-11) constructs as *ptr++, *++ptr, *ptr--, *--ptr. By the way, this is one of hard to memorize C feature. This is how it is:

C term	ARM term	C statement	how it works
Post-increment	post-indexed addressing	`*ptr++`	use `*ptr` value, then increment `ptr` pointer
Post-decrement	post-indexed addressing	`*ptr--`	use `*ptr` value, then decrement `ptr` pointer
Pre-increment	pre-indexed addressing	`*++ptr`	increment `ptr` pointer, then use `*ptr` value
Pre-decrement	post-indexed addressing	`*--ptr`	decrement `ptr` pointer, then use `*ptr` value

Dennis Ritchie (one of C language creators) mentioned that it's, probably, was invented by Ken Tompson (another C creator) because this processor feature was present in PDP-7 [Rit86] [Rit93]. Thus, C language compilers may use it, if it's present in targer processor.

Then one may spot CMP and BNE in loop body, these instructions continue operation until 0 will be met in string.

MVNS[51] (inverting all bits, NOT in x86 analogue) instructions and ADD computes $eos - str - 1$. In fact, these two instructions computes $R0 = str + eos$, which is equivalent to what was in source code, and why it's so, I already described here 1.11.1.

Apparently, LLVM, just like GCC, concludes this code will be shorter, or faster.

Optimizing Keil + ARM mode

Listing 1.34: Optimizing Keil + ARM mode

```
_strlen
                MOV     R1, R0

loc_2C8                                 ; CODE XREF: _strlen+14
                LDRB    R2, [R1],#1
                CMP     R2, #0
                SUBEQ   R0, R1, R0
                SUBEQ   R0, R0, #1
                BNE     loc_2C8
                BX      LR
```

Almost the same what we saw before, with the exception that $str - eos - 1$ expression may be computed not at the function's end, but right in loop body. -EQsuffix, as we may recall, mean that instruction will be executed only if operands in executed before CMP were equal to each other. Thus, if 0 will be in R0, both SUBEQ instructions are to be executed and result is leaving in R0.

[51] MoVe Not

1.12 Division by 9

Very simple function:

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

Is compiled in a very predictable way:

Listing 1.35: MSVC

```
_a$ = 8             ; size = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cdq            ; sign extend EAX to EDX:EAX
    mov    ecx, 9
    idiv   ecx
    pop    ebp
    ret    0
_f  ENDP
```

IDIV divides 64-bit number stored in EDX:EAX register pair by value in ECX register. As a result, EAX will contain quotient[52], and EDX — remainder. Result is returning from f() function in EAX register, so, that value is not moved anymore after division operation, it is in right place already. Because IDIV require value in EDX:EAX register pair, CDQ instruction (before IDIV) extending EAX value to 64-bit value taking value sign into account, just as MOVSX 1.11.1 does. If we turn optimization on (/Ox), we got:

Listing 1.36: Optimizing MSVC

```
_a$ = 8                     ; size = 4
_f    PROC

    mov    ecx, DWORD PTR _a$[esp-4]
    mov    eax, 954437177   ; 38e38e39H
    imul   ecx
    sar    edx, 1
    mov    eax, edx
    shr    eax, 31          ; 0000001fH
    add    eax, edx
    ret    0
_f    ENDP
```

This is — division using multiplication. Multiplication operation working much faster. And it is possible to use that trick [53] to produce a code which is equivalent and faster. GCC 4.4.1 even without optimization turned on, generate almost the same code as MSVC with optimization turned on:

Listing 1.37: Non-optimizing GCC 4.4.1

```
        public f
f       proc near

arg_0   = dword ptr  8

        push    ebp
        mov     ebp, esp
        mov     ecx, [ebp+arg_0]
        mov     edx, 954437177
        mov     eax, ecx
        imul    edx
```

[52] result of division

[53] Read more about division by multiplication in [War02, chapter 10] and: MSDN: Integer division by constants, http://www.nynaeve.net/?p=115

```
        sar     edx, 1
        mov     eax, ecx
        sar     eax, 1Fh
        mov     ecx, edx
        sub     ecx, eax
        mov     eax, ecx
        pop     ebp
        retn
f       endp
```

1.12.1 ARM

ARM processor, just like in any other ”pure” RISC-processors, lacks division instruction It lacks also a single instruction for multiplication by 32-bit constant. By using one clever trick, it’s possible to do division using only three instructions: addition, subtraction and bit shifts 1.15.

Here is an example of 32-bit number division by 10 from [Ltd94, 3.3 Division by a Constant]. Quotient and remainder on output.

```
; takes argument in a1
; returns quotient in a1, remainder in a2
; cycles could be saved if only divide or remainder is required
    SUB    a2, a1, #10             ; keep (x-10) for later
    SUB    a1, a1, a1, lsr #2
    ADD    a1, a1, a1, lsr #4
    ADD    a1, a1, a1, lsr #8
    ADD    a1, a1, a1, lsr #16
    MOV    a1, a1, lsr #3
    ADD    a3, a1, a1, asl #2
    SUBS   a2, a2, a3, asl #1      ; calc (x-10) - (x/10)*10
    ADDPL  a1, a1, #1              ; fix-up quotient
    ADDMI  a2, a2, #10             ; fix-up remainder
    MOV    pc, lr
```

Optimizing Xcode (LLVM) + ARM mode

```
__text:00002C58 39 1E 08 E3 E3 18 43 E3          MOV             R1, 0x38E38E39
__text:00002C60 10 F1 50 E7                      SMMUL           R0, R0, R1
__text:00002C64 C0 10 A0 E1                      MOV             R1, R0,ASR#1
__text:00002C68 A0 0F 81 E0                      ADD             R0, R1, R0,LSR#31
__text:00002C6C 1E FF 2F E1                      BX              LR
```

This code is mostly the same to what was generated by optimizing MSVC and GCC. Apparently, LLVM use the same algorithm for constants generating.

Observant reader may ask, how `MOV` writes 32-bit value in register, while this isn’t possible in ARM mode. It’s not possible indeed, but, as we see, there are 8 bytes per instruction instead of standard 4, in fact, there are two instructions. First instruction loading $0x8E39$ value into low 16 bit of register and second instruction is in fact `MOVT`, it loading $0x383E$ into high 16-bit of register. IDA 5 is aware of such sequences, and for the sake of compactness, reduced it to one single “pseudo-instruction”.

`SMMUL` (*Signed Most Significant Word Multiply*) instruction multilpy numbers treating them as signed numbers, and leaving high 32-bit part of result in `R0`, dropping low 32-bit part of result.

‘‘`MOV R1, R0,ASR#1`’’ instruction is arithmetic shift right by one bit.

‘‘`ADD R0, R1, R0,LSR#31`’’ is $R0 = R1 + R0 >> 31$

As a matter of fact, there are no separate shifting instruction in ARM mode. Instead, some instructions like (`MOV`, `ADD`, `SUB`, `RSB`)[54] May be supplied by option, is the second operand should be shifted, if yes, by what value and how. `ASR` meaning *Arithmetic Shift Right*, `LSR` — *Logican Shift Right*.

[54] These instructions are also called “data processing instructions”

Optimizing Xcode (LLVM) + thumb-2 mode

```
MOV             R1, 0x38E38E39
SMMUL.W         R0, R0, R1
ASRS            R1, R0, #1
ADD.W           R0, R1, R0,LSR#31
BX              LR
```

There are separate instructions for shifting in thumb mode, and one of them is used here — ASRS (arithmetic shift right).

Non-optimizing Xcode (LLVM) and Keil

Non-optimizing LLVM do not generate code we saw before in this section, but inserting a call to library function ___*divsi3* instead.

What about Keil: it inserting call to library function *__aeabi_idivmod* in all cases.

1.13 Work with FPU

FPU (*Floating-point unit*) — is a device within main CPU specially designed to work with floating point numbers.

It was called coprocessor in past. It looks like programmable calculator in some way and stay aside of main processor.

It is worth to study stack machines[55] before FPU studying, or learn Forth language basics[56].

It is interesting to know that in past (before 80486 CPU) coprocessor was a separate chip and it was not always settled on motherboard. It was possible to buy it separately and install.

Starting at 80486 CPU, FPU is always present in it.

FPU has a stack capable to hold 8 80-bit registers, each register can hold a number in IEEE 754[57] format.

C/C++ language offer at least two floating number types, *float* (*single-precision*[58], 32 bits) [59] and *double* (*double-precision*[60], 64 bits).

GCC also supports *long double* type (*extended precision*[61], 80 bit) but MSVC is not.

float type require the same number of bits as *int* type in 32-bit environment, but number representation is completely different.

Number consisting of sign, significand (also called *fraction*) and exponent.

Function having *float* or *double* among argument list is getting that value via stack. If function is returning *float* or *double* value, it leave that value in `ST(0)` register — at top of FPU stack.

1.13.1 Simple example

Let's consider simple example:

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

x86

Compile it in MSVC 2010:

Listing 1.38: 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]

; current stack state: ST(0) = _a

    fdiv   QWORD PTR __real@40091eb851eb851f

; current stack state: ST(0) = result of _a divided by 3.13
```

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

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

[57] http://en.wikipedia.org/wiki/IEEE_754-2008

[58] http://en.wikipedia.org/wiki/Single-precision_floating-point_format

[59] single precision float numbers format is also addressed in *Working with the float type as with a structure* 1.16.6 section

[60] http://en.wikipedia.org/wiki/Double-precision_floating-point_format

[61] http://en.wikipedia.org/wiki/Extended_precision

```
    fld    QWORD PTR _b$[ebp]

; current stack state: ST(0) = _b; ST(1) = result of _a divided by 3.13

    fmul   QWORD PTR __real@4010666666666666

; current stack state: ST(0) = result of _b * 4.1; ST(1) = result of _a divided by 3.13

    faddp  ST(1), ST(0)

; current stack state: ST(0) = result of addition

    pop    ebp
    ret    0
_f  ENDP
```

FLD takes 8 bytes from stack and load the number into ST(0) register, automatically converting it into internal 80-bit format *extended precision*).

FDIV divide value in register ST(0) by number storing at address __real@40091eb851eb851f — 3.14 value is coded there. Assembler syntax missing floating point numbers, so, what we see here is hexadecimal representation of *3.14* number in 64-bit IEEE 754 encoded.

After FDIV execution, ST(0) will hold quotient[62].

By the way, there are also FDIVP instruction, which divide ST(1) by ST(0), popping both these values from stack and then pushing result. If you know Forth language[63], you will quickly understand that this is stack machine[64].

Next FLD instruction pushing *b* value into stack.

After that, quotient is placed to ST(1), and ST(0) will hold *b* value.

Next FMUL instruction do multiplication: *b* from ST(0) register by value at __real@4010666666666666 (4.1 number is there) and leaves result in ST(0).

Very last FADDP instruction adds two values at top of stack, placing result at ST(1) register and then popping value at ST(1), hereby leaving result at top of stack in ST(0).

The function must return result in ST(0) register, so, after FADDP there are no any other code except of function epilogue.

GCC 4.4.1 (with -O3 option) emitting the same code, however, slightly different:

Listing 1.39: Optimizing GCC 4.4.1

```
                public f
f               proc near

arg_0           = qword ptr  8
arg_8           = qword ptr  10h

                push    ebp
                fld     ds:dbl_8048608 ; 3.14

; stack state now: ST(0) = 3.13

                mov     ebp, esp
                fdivr   [ebp+arg_0]

; stack state now: ST(0) = result of division

                fld     ds:dbl_8048610 ; 4.1

; stack state now: ST(0) = 4.1, ST(1) = result of division

                fmul    [ebp+arg_8]
```

[62] result of division

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

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

```
; stack state now: ST(0) = result of multiplication, ST(1) = result of division

                pop     ebp
                faddp   st(1), st

; stack state now: ST(0) = result of addition

                retn
f               endp
```

The difference is that, first of all, 3.14 is pushing to stack (into `ST(0)`), and then value in `arg_0` is dividing by what is in `ST(0)` register.

`FDIVR` meaning *Reverse Divide* — to divide with divisor and dividend swapped with each other. There are no such instruction for multiplication, because multiplication is commutative operation, so we have just `FMUL` without its `-R` counterpart.

`FADDP` adding two values but also popping one value from stack. After that operation, `ST(0)` will contain sum.

This fragment of disassembled code was produced using IDA 5 which named `ST(0)` register as `ST` for short.

ARM: Optimizing Xcode (LLVM) + ARM mode

Until ARM has floating standardized point support, several processor manufacturers may add their own instructions extensions. Then, VFP (*Vector Floating Point*) was standardized.

One important difference from x86, there you working with FPU-stack, but here, in ARM, there are no any stack, you work just with registers.

```
f
                VLDR            D16, =3.14
                VMOV            D17, R0, R1 ; load a
                VMOV            D18, R2, R3 ; load b
                VDIV.F64        D16, D17, D16 ; a/3.14
                VLDR            D17, =4.1
                VMUL.F64        D17, D18, D17 ; b*4.1
                VADD.F64        D16, D17, D16 ; +
                VMOV            R0, R1, D16
                BX              LR

dbl_2C98        DCFD 3.14               ; DATA XREF: f
dbl_2CA0        DCFD 4.1                ; DATA XREF: f+10
```

So, we see here new registers used, with D prefix. These are 64-bit registers, there are 32 of them, and these can be used both for floating-point numbers (double), but also for SIMD (it's called NEON here in ARM).

There are also 32 32-bit S-registers, they are intended to be used for single precision floating pointer numbers (float).

It's easy to remember: D-registers are intended for double precision numbers, while S-registers — for single precision numbers.

Both (3.14 and 4.1) constants are stored in memory in IEEE 754 form.

`VLDR` and `VMOV` instructions, as it can be easily deduced, are analogous to `LDR` and `MOV`, but they works with D-registers. It should be noted that these instructions, just like D-registers, are intended not only for floating point numbers, but can be also used for SIMD (NEON) operations and this will also be revealed soon.

Arguments are passed to function by usual way, via R-registers, however, each number having double precision has size 64-bits, so, for passing each, two R-registers are needed.

''`VMOV D17, R0, R1`'' at the very beginning, composing two 32-bit values from `R0` and `R1` into one 64-bit value and saves it to `D17`.

''`VMOV R0, R1, D16`'' is inverse operation, what was in `D16` leaving in two `R0` and `R1` registers, because, double-precision number, needing 64 bits for storage, returning in `R0` and `R1` registers pair.

`VDIV`, `VMUL` and `VADD`, are instruction for floating point numbers processing, computing, quotient[65], prod-

[65] result of division

uct[66] and sum[67], respectively.

The code for thumb-2 is same.

ARM: Optimizing Keil + thumb mode

```
f
                PUSH    {R3-R7,LR}
                MOVS    R7, R2
                MOVS    R4, R3
                MOVS    R5, R0
                MOVS    R6, R1
                LDR     R2, =0x66666666
                LDR     R3, =0x40106666
                MOVS    R0, R7
                MOVS    R1, R4
                BL      __aeabi_dmul
                MOVS    R7, R0
                MOVS    R4, R1
                LDR     R2, =0x51EB851F
                LDR     R3, =0x40091EB8
                MOVS    R0, R5
                MOVS    R1, R6
                BL      __aeabi_ddiv
                MOVS    R2, R7
                MOVS    R3, R4
                BL      __aeabi_dadd
                POP     {R3-R7,PC}

dword_364       DCD 0x66666666          ; DATA XREF: f+A
dword_368       DCD 0x40106666          ; DATA XREF: f+C
dword_36C       DCD 0x51EB851F          ; DATA XREF: f+1A
dword_370       DCD 0x40091EB8          ; DATA XREF: f+1C
```

Keil generates for processors not supporting FPU or NEON. So, double-precision floating numbers are passed via usual R-registers, and instead of FPU-instructions, service library functions are called, like `__aeabi_dmul`, `__aeabi_ddiv`, `__aeabi_dadd`, which emulating multiplication, division and addition floating-point numbers. Of course, that's slower than FPU-coprocessor, but it's better than nothing.

By the way, similar FPU-emulating libraries were very popular in x86 world when coprocessors were rare and expensive, and were installed only on expensive computers.

FPU-coprocessor emulating calling *soft float* or *armel* in ARM world, while coprocessor's FPU-instructions usage calling *hard float* or *armhf*.

For example, Linux kernel for Raspberry Pi is compiled in two variants. In *soft float* case, arguments will be passed via R-registers, and in *hard float* case — via D-registers.

And that's what don't let you use, for example, armhf-libraries from armel-code or vice versa, so that's why all code in Linux distribution should be compiled according to chosen calling convention.

1.13.2 Passing floating point number via arguments

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

        return 0;
}
```

[66] result of multiplication

[67] result of addition

x86

Let's see what we got in (MSVC 2010):

Listing 1.40: 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  ; allocate place for the first variable
    fld    QWORD PTR __real@3ff8a3d70a3d70a4
    fstp   QWORD PTR [esp]
    sub    esp, 8  ; allocate place for the second variable
    fld    QWORD PTR __real@40400147ae147ae1
    fstp   QWORD PTR [esp]
    call   _pow
    add    esp, 8  ; "return back" place of one variable.

; in local stack here 8 bytes still reserved for us.
; result now in ST(0)

    fstp   QWORD PTR [esp] ; move result from ST(0) to local stack for printf()
    push   OFFSET $SG2651
    call   _printf
    add    esp, 12
    xor    eax, eax
    pop    ebp
    ret    0
_main    ENDP
```

FLD and FSTP are moving variables from/to data segment to FPU stack. `pow()`[68] taking both values from FPU-stack and returning result in `ST(0)`. `printf()` takes 8 bytes from local stack and interpret them as *double* type variable.

ARM + Non-optimizing Xcode (LLVM) + thumb-2 mode

```
_main

var_C           = -0xC

                PUSH            {R7,LR}
                MOV             R7, SP
                SUB             SP, SP, #4
                VLDR            D16, =32.01
                VMOV            R0, R1, D16
                VLDR            D16, =1.54
                VMOV            R2, R3, D16
                BLX             _pow
                VMOV            D16, R0, R1
                MOV             R0, 0xFC1 ; "32.01 ^ 1.54 = %lf\n"
                ADD             R0, PC
                VMOV            R1, R2, D16
                BLX             _printf
                MOVS            R1, 0
                STR             R0, [SP,#0xC+var_C]
                MOV             R0, R1
                ADD             SP, SP, #4
                POP             {R7,PC}
```

[68] standard C function, raises a number to the given power

```
dbl_2F90        DCFD 32.01              ; DATA XREF: _main+6
dbl_2F98        DCFD 1.54               ; DATA XREF: _main+E
```

As I wrote before, 64-bit floating pointer numbers passing in R-registers pairs. This is code is redundant for a little (certainty because optimization is turned off), because, it's actually possible to load values into R-registers straightforwardly without touching D-registers.

So, as we see, `_pow` function receiving first argument in `R0` and `R1`, and the second one in `R2` and `R3`. Function leave result in `R0` and `R1`. Result of `_pow` is moved into `D16`, then in `R1` and `R2` pair, from where `printf()` will take this number.

ARM + Non-optimizing Keil + ARM mode

```
_main
                STMFD   SP!, {R4-R6,LR}
                LDR     R2, =0xA3D70A4  ; y
                LDR     R3, =0x3FF8A3D7
                LDR     R0, =0xAE147AE1 ; x
                LDR     R1, =0x40400147
                BL      pow
                MOV     R4, R0
                MOV     R2, R4
                MOV     R3, R1
                ADR     R0, a32_011_54Lf ; "32.01 ^ 1.54 = %lf\n"
                BL      __2printf
                MOV     R0, #0
                LDMFD   SP!, {R4-R6,PC}

y               DCD 0xA3D70A4           ; DATA XREF: _main+4
dword_520       DCD 0x3FF8A3D7          ; DATA XREF: _main+8
; double x
x               DCD 0xAE147AE1          ; DATA XREF: _main+C
dword_528       DCD 0x40400147          ; DATA XREF: _main+10
a32_011_54Lf    DCB "32.01 ^ 1.54 = %lf",0xA,0
                                        ; DATA XREF: _main+24
```

D-registers are not used here, only R-register pairs are used.

1.13.3 Comparison example

Let's try this:

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

        return b;
};
```

x86

Despite simplicity of that function, it will be harder to understand how it works.

MSVC 2010 generated:

Listing 1.41: 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]

; current stack state: ST(0) = _b
; compare _b (ST(0)) and _a, and pop register

    fcomp  QWORD PTR _a$[ebp]

; stack is empty here

    fnstsw ax
    test   ah, 5
    jp     SHORT $LN1@d_max

; we are here only 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
```

So, FLD loading _b into ST(0) register.

FCOMP compares ST(0) register state with what is in _a value and set C3/C2/C0 bits in FPU status word register. This is 16-bit register reflecting current state of FPU.

For now C3/C2/C0 bits are set, but unfortunately, CPU before Intel P6 [69] hasn't any conditional jumps instructions which are checking these bits. Probably, it is a matter of history (remember: FPU was separate chip in past). Modern CPU starting at Intel P6 has FCOMI/FCOMIP/FUCOMI/FUCOMIP instructions — which does that same, but modifies CPU flags ZF/PF/CF.

After bits are set, the FCOMP instruction popping one variable from stack. This is what distinguish it from FCOM, which is just comparing values, leaving the stack at the same state.

FNSTSW copies FPU status word register to AX. Bits C3/C2/C0 are placed at positions 14/10/8, they will be at the same positions in AX registers and all them are placed in high part of AX — AH.

- If b>a in our example, then C3/C2/C0 bits will be set as following: 0, 0, 0.
- If a>b, then bits will be set: 0, 0, 1.
- If a=b, then bits will be set: 1, 0, 0.

After `test ah, 5` execution, bits C3 and C1 will be set to 0, but at positions 0 and 2 (in AH registers) C0 and C2 bits will be leaved.

Now let's talk about parity flag. Another notable epoch rudiment:

> 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.[70]

This flag is to be set to 1 if ones number is even. And to zero if odd.

[69] Intel P6 is Pentium Pro, Pentium II, etc

Thus, PF flag will be set to 1 if both C0 and C2 are set to zero or both are ones. And then following JP (*jump if PF==1*) will be triggered. If we recall values of C3/C2/C0 for different cases, we will see that conditional jump JP will be triggered in two cases: if b>a or a==b (C3 bit is already not considering here, because it was cleared while execution of `test ah, 5` instruction).

It is all simple thereafter. If conditional jump was triggered, FLD will load _b value to ST(0), and if it's not triggered, _a will be loaded.

But it is not over yet!

Now let's compile it with MSVC 2010 with optimization option /Ox

Listing 1.42: Optimizing MSVC 2010

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

; current stack state: ST(0) = _a, ST(1) = _b

    fcom    ST(1) ; compare _a and ST(1) = (_b)
    fnstsw    ax
    test    ah, 65                    ; 00000041H
    jne    SHORT $LN5@d_max
    fstp    ST(1)  ; copy ST(0) to ST(1) and pop register, leave (_a) on top

; current stack state: ST(0) = _a

    ret    0
$LN5@d_max:
    fstp    ST(0)  ; copy ST(0) to ST(0) and pop register, leave (_b) on top

; current stack state: ST(0) = _b

    ret    0
_d_max    ENDP
```

FCOM is distinguished from FCOMP is that sense that it just comparing values and leave FPU stack in the same state. Unlike previous example, operands here in reversed order. And that is why result of comparision in C3/C2/C0 will be different:

- If a>b in our example, then C3/C2/C0 bits will be set as: 0, 0, 0.
- If b>a, then bits will be set as: 0, 0, 1.
- If a=b, then bits will be set as: 1, 0, 0.

It can be said, `test ah, 65` instruction just leave two bits — C3 and C0. Both will be zeroes if a>b: in that case JNE jump will not be triggered. Then `FSTP ST(1)` is following — this instruction copies ST(0) value into operand and popping one value from FPU stack. In other words, that instruction copies ST(0) (where _a value now) into ST(1). After that, two values of _a are at the top of stack now. After that, one value is popping. After that, ST(0) will contain _a and function is finished.

Conditional jump JNE is triggered in two cases: of b>a or a==b. ST(0) into ST(0) will be copied, it is just like idle (NOP) operation, then one value is popping from stack and top of stack (ST(0)) will contain what was in ST(1) before (that is _b). Then function finishes. That instruction used here probably because FPU has no instruction to pop value from stack and not to store it anywhere.

Well, but it is still not over.

GCC 4.4.1

Listing 1.43: 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

; put a and b to local stack:

    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

; load a and b to FPU stack:

    fld     [ebp+a]
    fld     [ebp+b]

; current stack state: ST(0) - b; ST(1) - a

    fxch    st(1) ; this instruction swapping ST(1) and ST(0)

; current stack state: ST(0) - a; ST(1) - b

    fucompp    ; compare a and b and pop two values from stack, i.e., a and b
    fnstsw  ax ; store FPU status to AX
    sahf       ; load SF, ZF, AF, PF, and CF flags state from AH
    setnbe  al ; store 1 to AL if CF=0 and ZF=0
    test    al, al            ; AL==0 ?
    jz      short loc_8048453 ; yes
    fld     [ebp+a]
    jmp     short locret_8048456

loc_8048453:
    fld     [ebp+b]

locret_8048456:
    leave
    retn
d_max endp
```

FUCOMPP — is almost like FCOM, but popping both values from stack and handling "not-a-numbers" differently.

More about *not-a-numbers*:

FPU is able to work with special values which are *not-a-numbers* or NaNs[71]. These are infinity, result of dividing by zero, etc. Not-a-numbers can be "quiet" and "signalling". It is possible to continue to work with "quiet" NaNs, but if one try to do some operation with "signalling" NaNs — an exception will be raised.

FCOM will raise exception if any operand — NaN. FUCOM will raise exception only if any operand — signalling NaN (SNaN).

[71] http://en.wikipedia.org/wiki/NaN

The following instruction is SAHF — this is rare instruction in the code which is not use FPU. 8 bits from AH is movinto into lower 8 bits of CPU flags in the following order: SF:ZF:-:AF:-:PF:-:CF <- AH.

Let's remember that FNSTSW is moving interesting for us bits C3/C2/C0 into AH and they will be in positions 6, 2, 0 in AH register.

In other words, fnstsw ax / sahf instruction pair is moving C3/C2/C0 into ZF, PF, CF CPU flags.

Now let's also recall, what values of C3/C2/C0 bits will be set:

- If a is greater than b in our example, then C3/C2/C0 bits will be set as: 0, 0, 0.
- if a is less than b, then bits will be set as: 0, 0, 1.
- If a=b, then bits will be set: 1, 0, 0.

In other words, after FUCOMPP/FNSTSW/SAHF instructions, we will have these CPU flags states:

- If a>b, CPU flags will be set as: ZF=0, PF=0, CF=0.
- If a<b, then CPU flags will be set as: ZF=0, PF=0, CF=1.
- If a=b, then CPU flags will be set as: ZF=1, PF=0, CF=0.

How SETNBE instruction will store 1 or 0 to AL: it is depends of CPU flags. It is almost JNBE instruction counterpart, with exception that SETcc[72] is storing 1 or 0 to AL, but Jcc do actual jump or not. SETNBE store 1 only if CF=0 and ZF=0. If it is not true, zero will be stored into AL.

Both CF is 0 and ZF is 0 simultaneously only in one case: if a>b.

Then one will be stored to AL and the following JZ will not be triggered and function will return _a. On all other cases, _b will be returned.

But it is still not over.

GCC 4.4.1 with -O3 optimization turned on

Listing 1.44: Optimizing GCC 4.4.1

```
                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

; stack state now: ST(0) = _b, ST(1) = _a
                fxch    st(1)

; stack state now: ST(0) = _a, ST(1) = _b
                fucom   st(1) ; compare _a and _b
                fnstsw  ax
                sahf
                ja      short loc_8048448

; store ST(0) to ST(0) (idle operation), pop value at top of stack, leave _b at top
                fstp    st
                jmp     short loc_804844A

loc_8048448:
; store _a to ST(0), pop value at top of stack, leave _a at top
                fstp    st(1)
```

[72] *cc* is *condition code*

```
loc_804844A:
                pop     ebp
                retn
d_max           endp
```

It is almost the same except one: JA usage instead of SAHF. Actually, conditional jump instructions checking "larger", "lesser" or "equal" for unsigned number comparison (JA, JAE, JBE, JBE, JE/JZ, JNA, JNAE, JNB, JNBE, JNE/JNZ) are checking only CF and ZF flags. And C3/C2/C0 bits after comparison are moving into these flags exactly in the same fashion so conditional jumps will work here. JA will work if both CF are ZF zero.

Thereby, conditional jumps instructions listed here can be used after FNSTSW/SAHF instructions pair.

It seems, FPU C3/C2/C0 status bits was placed there deliberately so to map them to base CPU flags without additional permutations.

ARM + Optimizing Xcode (LLVM) + ARM mode

Listing 1.45: Optimizing Xcode (LLVM) + ARM mode

```
VMOV            D16, R2, R3 ; b
VMOV            D17, R0, R1 ; a
VCMPE.F64       D17, D16
VMRS            APSR_nzcv, FPSCR
VMOVGT.F64      D16, D17 ; copy b to D16
VMOV            R0, R1, D16
BX              LR
```

A very simple case. Input values are placed into D17 and D16 and then compared with the help of VCMPE instruction. Just like in x86 coprocessor, ARM coprocessor has its own status and flags register, (FPSCR), because there is a need to store coprocessor-specific flags.

And just like in x86, there are no conditional jump instruction in ARM, checking bits in coprocessor status register, so there is VMRS instruction, copying 4 bits (N, Z, C, V) from the coprocessor status word into bits of *general* status (APSR register).

VMOVGT is analogue of MOVGT, instruction, to be executed if one operand is greater than other while comparing (*GT — Greater Than*).

If it will be executed, b value will be written into D16, stored at that moment in D17.

And if it will not be triggered, then a value will stay in D16.

Penultimate instruction VMOV will prepare value in D16 for returning via R0 and R1 registers pair.

ARM + Optimizing Xcode (LLVM) + thumb-2 mode

Listing 1.46: Optimizing Xcode (LLVM) + thumb-2 mode

```
VMOV            D16, R2, R3 ; b
VMOV            D17, R0, R1 ; a
VCMPE.F64       D17, D16
VMRS            APSR_nzcv, FPSCR
IT GT
VMOVGT.F64      D16, D17
VMOV            R0, R1, D16
BX              LR
```

Almost the same as in previous example, howeverm slightly different. As a matter of fact, many instructions in ARM mode can be supplied by condition predicate, and the instruction is to be executed if condition is true.

But there are no such thing in thumb mode. There are no place in 16-bit instructions for spare 4 bits where condition can be encoded.

However, thumb-2 was extended to make possible to specify predicates to old thumb instructions.

Here, is the IDA 5-generated listing, we see VMOVGT instruction, the same as in previous example.

But in fact, usual VMOV is encoded there, but IDA 5 added -GT suffix to it, because there are "IT GT" instruction placed right before.

IT instruction defines so-called *if-then block*. After that instruction, it's possible to place up to 4 instructions, to which predicate suffix will be added. In our example, ''IT GT'' meaning, the next instruction will be executed, if *GT* (*Greater Than*) condition is true.

Now more complex code fragment, by the way, from "Angry Birds" (for iOS):

Listing 1.47: Angry Birds Classic

```
ITE NE
VMOVNE          R2, R3, D16
VMOVEQ          R2, R3, D17
```

ITE meaning *if-then-else* and it encode suffixes for two next instructions. First instruction will execute if condition encoded in ITE (*NE, not equal*) will be true at that moment, and the second — if that condition will not be true. (Inverse condition of NE is EQ (*equal*)).

Slightly harder, and this fragment from "Angry Birds" as well:

Listing 1.48: Angry Birds Classic

```
ITTTT EQ
MOVEQ           R0, R4
ADDEQ           SP, SP, #0x20
POPEQ.W         {R8,R10}
POPEQ           {R4-R7,PC}
```

4 "T" symbols in instruction mnemonic mean that 4 next instructions will be executed if condition is true. That's why IDA 5 added -EQ suffix to each 4 instructions.

And if there will be, for example, ITEEE EQ (*if-then-else-else-else*), then suffixes will be set as follows:

```
-EQ
-NE
-NE
-NE
```

Another fragment from "Angry Birds":

Listing 1.49: Angry Birds Classic

```
CMP.W           R0, #0xFFFFFFFF
ITTE LE
SUBLE.W         R10, R0, #1
NEGLE           R0, R0
MOVGT           R10, R0
```

ITTE (*if-then-then-else*) mean that 1st and 2nd instructions will be executed, if LE (*Less or Equal*) condition is true, and 3rd — if inverse condition (GT — *Greater Than*) is true.

Compilers usually are not generating all possible combinations. For example, it mentioned "Angry Birds" game (*classic* version for iOS) only these cases of IT instruction are used: IT, ITE, ITT, ITTE, ITTT, ITTTT. How I learnt this? In IDA 5 it's possible to produce listing files, so I did it, but I also set in options to show 4 bytes of each opcodes . Then, knowing that high part of 16-bit opcode IT is $0xBF$, I did this with grep:

```
cat AngryBirdsClassic.lst | grep " BF" | grep "IT" > results.lst
```

By the way, if to program in ARM assembly language manually for thumb-2 mode, with adding conditional suffixes, assembler will add IT instructions automatically, with respectable flags, where it's need.

ARM + Non-optimizing Xcode (LLVM) + ARM mode

Listing 1.50: Non-optimizing Xcode (LLVM) + ARM mode

```
b               = -0x20
a               = -0x18
val_to_return   = -0x10
saved_R7        = -4
```

```
                STR             R7, [SP,#saved_R7]!
                MOV             R7, SP
                SUB             SP, SP, #0x1C
                BIC             SP, SP, #7
                VMOV            D16, R2, R3
                VMOV            D17, R0, R1
                VSTR            D17, [SP,#0x20+a]
                VSTR            D16, [SP,#0x20+b]
                VLDR            D16, [SP,#0x20+a]
                VLDR            D17, [SP,#0x20+b]
                VCMPE.F64       D16, D17
                VMRS            APSR_nzcv, FPSCR
                BLE             loc_2E08
                VLDR            D16, [SP,#0x20+a]
                VSTR            D16, [SP,#0x20+val_to_return]
                B               loc_2E10

loc_2E08
                VLDR            D16, [SP,#0x20+b]
                VSTR            D16, [SP,#0x20+val_to_return]

loc_2E10
                VLDR            D16, [SP,#0x20+val_to_return]
                VMOV            R0, R1, D16
                MOV             SP, R7
                LDR             R7, [SP+0x20+b],#4
                BX              LR
```

Almost the same we already saw, but too much redundant code because of a and b variables storage in local stack, as well as returning value.

ARM + Optimizing Keil + thumb mode

Listing 1.51: Optimizing Keil + thumb mode

```
                PUSH    {R3-R7,LR}
                MOVS    R4, R2
                MOVS    R5, R3
                MOVS    R6, R0
                MOVS    R7, R1
                BL      __aeabi_cdrcmple
                BCS     loc_1C0
                MOVS    R0, R6
                MOVS    R1, R7
                POP     {R3-R7,PC}

loc_1C0
                MOVS    R0, R4
                MOVS    R1, R5
                POP     {R3-R7,PC}
```

Keil not generate special instruction for float numbers comparing, because, it cannot rely it will be supported on the target CPU, and it cannot be done by simple bitwise comparing. So there is called external library function for comparing: `__aeabi_cdrcmple`. Please note, comparision result is to be leaved in flags, so the following `BCS` (*Carry set - Greater than or equal*) instruction may work without any additional code.

1.14 Arrays

Array is just a set of variables in memory, always lying next to each other, always has same type.

1.14.1 Simple example

```
#include <stdio.h>

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

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

        for (i=0; i<20; i++)
                printf ("a[%d]=%d\n", i, a[i]);

        return 0;
};
```

x86

Let's compile:

Listing 1.52: MSVC

```
_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
```

Nothing very special, just two loops: first is filling loop and second is printing loop. `shl ecx, 1` instruction is used for ECX value multiplication by 2, more about below 1.15.3.

80 bytes are allocated in stack for array, that's 20 elements of 4 bytes.

Here is what GCC 4.4.1 does:

Listing 1.53: 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
```

By the way, *a* variable has *int** type (that is pointer to *int*) — you can try to pass a pointer to array to another function, but it much correctly to say that pointer to the first array elemnt is passed (addresses of another element's places are calculated in obvious way). If to index this pointer as *a[idx]*, *idx* just to be added to the pointer and the element placed there (to which calculated pointer is pointing) returned.

An interesting example: string of characters like *"string"* is array of characters and it has *const char** type.Index can be applied to this pointer. And that's why it's possible to write like ‘‘`string`’’`[i]` — this is correct C/C++ expression!

ARM + Non-optimizing Keil + ARM mode

```
                EXPORT _main
_main
                STMFD   SP!, {R4,LR}
                SUB     SP, SP, #0x50      ; allocate place for 20 int variables

; first loop

                MOV     R4, #0             ; i
                B       loc_4A0
loc_494
                MOV     R0, R4,LSL#1       ; R0=R4*2
                STR     R0, [SP,R4,LSL#2]  ; store R0 to SP+R4<<2 (same as SP+R4*4)
                ADD     R4, R4, #1         ; i=i+1

loc_4A0
                CMP     R4, #20            ; i<20?
                BLT     loc_494            ; yes, run loop body again

; second loop

                MOV     R4, #0             ; i
                B       loc_4C4
loc_4B0
                LDR     R2, [SP,R4,LSL#2]  ; (second printf argument) R2=*(SP+R4<<4) (same as *(SP+R4*4))
                MOV     R1, R4             ; (first printf argument) R1=i
                ADR     R0, aADD           ; "a[%d]=%d\n"
                BL      __2printf
                ADD     R4, R4, #1         ; i=i+1

loc_4C4
                CMP     R4, #20            ; i<20?
                BLT     loc_4B0            ; yes, run loop body again
                MOV     R0, #0             ; value to return
                ADD     SP, SP, #0x50      ; deallocate place for 20 int variables
                LDMFD   SP!, {R4,PC}
```

int type require 32 bits for storage, or 4 bytes, so for storage of 20 *int* variables, 80 ($0x50$) bytes are needed, so that's why ‘‘`SUB SP, SP, #0x50`’’ instruction in function epilogue allocates exactly this ammount of space in local stack.

In both first and second loops, i loop iterator will be placed in `R4` register.

A number to be written into array, is calculating as $i * 2$ which is equivalent to shifting left by one bit, so ‘‘`MOV R0, R4,LSL#1`’’ instruction do this.

‘‘`STR R0, [SP,R4,LSL#2]`’’ writes `R0` contents into array. Here is how a pointer to array element is to be calculated: `SP` pointing to array begin, `R4` is i. So shift i left by 2 bits, that's equivalent to multiplication by 4 (because each array element has size of 4 bytes) and add it to address of array begin.

The second loop has inverse ‘‘`LDR R2, [SP,R4,LSL#2]`’’, instruction, it loads from array value we need, and the pointer to it is calculated in exactly the same way.

ARM + Optimizing Keil + thumb mode

```
_main
                PUSH    {R4,R5,LR}
                SUB     SP, SP, #0x54   ; allocate place for 20 int variables + one more variable

; first loop

                MOVS    R0, #0          ; i
                MOV     R5, SP          ; pointer to first array element

loc_1CE
                LSLS    R1, R0, #1      ; R1=i<<1 (same as i*2)
                LSLS    R2, R0, #2      ; R2=i<<2 (same as i*4)
                ADDS    R0, R0, #1      ; i=i+1
                CMP     R0, #20         ; i<20?
                STR     R1, [R5,R2]     ; store R1 to *(R5+R2) (same R5+i*4)
                BLT     loc_1CE         ; yes, i<20, run loop body again

; second loop

                MOVS    R4, #0          ; i=0
loc_1DC
                LSLS    R0, R4, #2      ; R0=i<<2 (same as i*4)
                LDR     R2, [R5,R0]     ; load from *(R5+R0) (same as R5+i*4)
                MOVS    R1, R4
                ADR     R0, aADD        ; "a[%d]=%d\n"
                BL      __2printf
                ADDS    R4, R4, #1      ; i=i+1
                CMP     R4, #20         ; i<20?
                BLT     loc_1DC         ; yes, i<20, run loop body again
                MOVS    R0, #0          ; value to return
                ADD     SP, SP, #0x54   ; deallocate place for 20 int variables + one more variable
                POP     {R4,R5,PC}
```

Thumb code is very similar. Thumb mode has special instructions for bit shifting (like LSLS), calculating value to be written into array and address of each element in array as well.

Compiler allocates slightly more space in local stack, however, last 4 bytes are not used.

1.14.2 Buffer overflow

So, array indexing is just *array[index]*. If you study generated code closely, you'll probably note missing index bounds checking, which could check index, *if it is less than 20*. What if index will be greater than 20? That's the one C/C++ feature it's often blamed for.

Here is a code successfully compiling and working:

```
#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;
};
```

Compilation results (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
```

I'm running it, and I got:

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

It is just *something*, occasionally lying in the stack near to array, 400 bytes from its first element.

Indeed, how it could be done differently? Compiler may incorporate some code, checking index value to be always in array's bound, like in higher-level programming languages[73], but this makes running code slower.

OK, we read some values in stack *illegally*, but what if we could write something to it?

Here is what we will write:

```
#include <stdio.h>

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

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

        return 0;
};
```

And what we've got:

```
_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
```

[73] Java, Python, etc

```
$LN2@main:
 mov    eax, DWORD PTR _i$[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$[ebp]      ; that insruction is obviously redundant
 mov    DWORD PTR _a$[ebp+ecx*4], edx ; ECX could be used as second operand here instead
 jmp    SHORT $LN2@main
$LN1@main:
 xor    eax, eax
 mov    esp, ebp
 pop    ebp
 ret    0
_main   ENDP
```

Run compiled program and its crashing. No wonder. Let's see, where exactly it's crashing.

I'm not using debugger anymore, because I tired to run it each time, move mouse, etc, when I need just to spot some register's state at specific point. That's why I wrote very minimalistic tool for myself, *tracer* 5.0.1, which is enough for my tasks.

I can also use it just to see, where debuggee is crashed. So let's see:

```
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
```

Now please keep your eyes on registers.

Exception occured at address 0x15. It's not legal address for code — at least for win32 code! We trapped there somehow against our will. It's also interesting fact that EBP register contain 0x14, ECX and EDX — 0x1D.

Let's study stack layout more.

After control flow was passed into `main()`, EBP register value was saved into stack. Then, 84 bytes was allocated for array and *i* variable. That's `(20+1)*sizeof(int)`. ESP pointing now to `_i` variable in local stack and after execution of next `PUSH something`, *something* will be appeared next to `_i`.

That's stack layout while control is inside `main()`:

ESP	4 bytes for *i*
ESP+4	80 bytes for `a[20]` array
ESP+84	saved EBP value
ESP+88	returning address

Instruction `a[19]=something` writes last *int* in array bounds (in bounds yet!)

Instruction `a[20]=something` writes *something* to the place where EBP value is saved.

Please take a look at registers state at the crash moment. In our case, number 20 was written to 20th element. By the function ending, function epilogue restore EBP value. (20 in decimal system is 0x14 in hexadecimal). Then, RET instruction was executed, which is equivalent to `POP EIP` instruction.

RET instruction taking returning adddress from stack (that's address in some CRT[74]-function, which was called `main()`), and 21 was stored there (0x15 in hexadecimal). The CPU trapped at the address 0x15, but there are no executable code, so exception was raised.

Welcome! It's called *buffer overflow*[75].

[74] C Run-Time

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

Replace *int* array by string (*char* array), create a long string deliberately, pass it to the program, to the function which is not checking string length and copies it to short buffer, and you'll able to point to a program an address to which it should jump. Not that simple in reality, but that's how it was emerged[76].

Let's try the same code in GCC 4.4.1. We got:

```
                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
loc_80483D1:
                cmp     [ebp+i], 1Dh
                jle     short loc_80483C3
                mov     eax, 0
                leave
                retn
main            endp
```

Running this in Linux will produce: `Segmentation fault`.

If we run this in GDB debugger, we getting this:

```
(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)
```

Register values are slightly different then in win32 example, that's because stack layout is slightly different too.

1.14.3 Buffer overflow protection methods

There are several methods to protect against it, regardless of C/C++ programmers' negligence. MSVC has options like[77]:

[76] Classic article about it: Smashing The Stack For Fun And Profit

[77] Wikipedia: compiler-side buffer overflow protection methods

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

One of the methods is to write random value among local variables to stack at function prologue and to check it in function epilogue before function exiting. And if value is not the same, do not execute last instruction RET, but halt (or hang). Process will hang, but that's much better then remote attack to your host.

This random value is called "canary" sometimes, it's related to miner's canary[78], they were used by miners in these days, in order to detect poisonous gases quickly. Canaries are very sensetive to mine gases, they become very agitated in case of danger, or even dead.

If to compile our very simple array example 1.14.1 in MSVC with RTC1 and RTCs option, you will see call to @_RTC_CheckStackVars@8 function at the function end, checking "canary" correctness.

Optimizing Xcode (LLVM) + thumb-2 mode

Let's back to our simple array example 1.14.1, again, now we can see how LLVM will check "canary" correctness:

```
_main

var_64          = -0x64
var_60          = -0x60
var_5C          = -0x5C
var_58          = -0x58
var_54          = -0x54
var_50          = -0x50
var_4C          = -0x4C
var_48          = -0x48
var_44          = -0x44
var_40          = -0x40
var_3C          = -0x3C
var_38          = -0x38
var_34          = -0x34
var_30          = -0x30
var_2C          = -0x2C
var_28          = -0x28
var_24          = -0x24
var_20          = -0x20
var_1C          = -0x1C
var_18          = -0x18
canary          = -0x14
var_10          = -0x10

                PUSH            {R4-R7,LR}
                ADD             R7, SP, #0xC
                STR.W           R8, [SP,#0xC+var_10]!
                SUB             SP, SP, #0x54
                MOVW            R0, #aObjc_methtype ; "objc_methtype"
                MOVS            R2, #0
                MOVT.W          R0, #0
                MOVS            R5, #0
                ADD             R0, PC
                LDR.W           R8, [R0]
                LDR.W           R0, [R8]
                STR             R0, [SP,#0x64+canary]
                MOVS            R0, #2
                STR             R2, [SP,#0x64+var_64]
                STR             R0, [SP,#0x64+var_60]
                MOVS            R0, #4
                STR             R0, [SP,#0x64+var_5C]
                MOVS            R0, #6
                STR             R0, [SP,#0x64+var_58]
                MOVS            R0, #8
                STR             R0, [SP,#0x64+var_54]
```

[78] Wikipedia: Miner's canary

```
                MOVS            R0, #0xA
                STR             R0, [SP,#0x64+var_50]
                MOVS            R0, #0xC
                STR             R0, [SP,#0x64+var_4C]
                MOVS            R0, #0xE
                STR             R0, [SP,#0x64+var_48]
                MOVS            R0, #0x10
                STR             R0, [SP,#0x64+var_44]
                MOVS            R0, #0x12
                STR             R0, [SP,#0x64+var_40]
                MOVS            R0, #0x14
                STR             R0, [SP,#0x64+var_3C]
                MOVS            R0, #0x16
                STR             R0, [SP,#0x64+var_38]
                MOVS            R0, #0x18
                STR             R0, [SP,#0x64+var_34]
                MOVS            R0, #0x1A
                STR             R0, [SP,#0x64+var_30]
                MOVS            R0, #0x1C
                STR             R0, [SP,#0x64+var_2C]
                MOVS            R0, #0x1E
                STR             R0, [SP,#0x64+var_28]
                MOVS            R0, #0x20
                STR             R0, [SP,#0x64+var_24]
                MOVS            R0, #0x22
                STR             R0, [SP,#0x64+var_20]
                MOVS            R0, #0x24
                STR             R0, [SP,#0x64+var_1C]
                MOVS            R0, #0x26
                STR             R0, [SP,#0x64+var_18]
                MOV             R4, 0xFDA ; "a[%d]=%d\n"
                MOV             R0, SP
                ADDS            R6, R0, #4
                ADD             R4, PC
                B               loc_2F1C

; second loop begin

loc_2F14
                ADDS            R0, R5, #1
                LDR.W           R2, [R6,R5,LSL#2]
                MOV             R5, R0

loc_2F1C
                MOV             R0, R4
                MOV             R1, R5
                BLX             _printf
                CMP             R5, #0x13
                BNE             loc_2F14
                LDR.W           R0, [R8]
                LDR             R1, [SP,#0x64+canary]
                CMP             R0, R1
                ITTTT EQ                    ; canary still correct?
                MOVEQ           R0, #0
                ADDEQ           SP, SP, #0x54
                LDREQ.W         R8, [SP+0x64+var_64],#4
                POPEQ           {R4-R7,PC}
                BLX             ___stack_chk_fail
```

First of all, as we see, LLVM made loop “unrolled” and all values are written into array one-by-one, already calculated, because LLVM conclued it will be faster. By the way, ARM mode instructions may help to do this even faster, and finding this way could be your homework.

At the function end wee see “canaries” comparison — that laying in local stack and correct one, to which R8 register pointing. If they are equal to each other, 4-instruction block is triggered by ‘‘ITTTT EQ’’, it is writing 0 into R0, function epilogue and exit. If “canaries” are not equial, block will not be triggered, and jump to

`___stack_chk_fail` function will be occured, which, as I suppose, will halt execution.

1.14.4 One more word about arrays

Now we understand, why it's not possible to write something like that in C/C++ code [79]:

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

That's just because compiler should know exact array size to allocate place for it in local stack layout or in data segment (in case of global variable) on compiling stage.

If you need array of arbitrary size, allocate it by `malloc()`, then access allocated memory block as array of variables of type you need. Or use C99 standard feature [ISO07, 6.7.5.2], but it will be looks like alloca() 1.2.3 internally.

1.14.5 Multidimensional arrays

Internally, multidimensional array is essentially the same thing as linear array.

Because computer memory in linear, it's one-dimensional array. But this one-dimensional array can be easily represented as multidimensional for convenience.

For example, that's how $a[3][4]$ array elements will be placed in one-dimensional array of 12 cells:

0	1	2	3
4	5	6	7
8	9	10	11

So, in order to address elements we need, first multiply first index by 4 (matrix width) and then add second index. That's called *row-major order*, and this method of arrays and matrices representation is used in at least in C/C++, Python. *row-major order* term in plain English language mean: "first, write elements of first row, then second row ... and finally elements of last row".

Another method of representation called *column-major order* (array indices used in reverse order) and it's used at least in FORTRAN, MATLAB, R. *column-major order* term in plain English language mean: "first, write elements of first column, then second columnt ... and finally elements of last column".

Same thing about multidimensional arrays.

Let's see:

Listing 1.54: simple example

```
#include <stdio.h>

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

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

x86

We got (MSVC 2010):

Listing 1.55: MSVC 2010

```
_DATA    SEGMENT
COMM     _a:DWORD:01770H
```

[79] However, it's possible in C99 standard [ISO07, 6.7.5.2]: GCC is actually do this by allocating array dynammically in stack (like alloca() 1.2.3)

```
_DATA    ENDS
PUBLIC   _insert
_TEXT    SEGMENT
_x$ = 8                 ; size = 4
_y$ = 12                ; size = 4
_z$ = 16                ; size = 4
_value$ = 20            ; size = 4
_insert    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _x$[ebp]
    imul   eax, 2400                    ; eax=600*4*x
    mov    ecx, DWORD PTR _y$[ebp]
    imul   ecx, 120                     ; ecx=30*4*y
    lea    edx, DWORD PTR _a[eax+ecx]   ; edx=a + 600*4*x + 30*4*y
    mov    eax, DWORD PTR _z$[ebp]
    mov    ecx, DWORD PTR _value$[ebp]
    mov    DWORD PTR [edx+eax*4], ecx   ; *(edx+z*4)=value
    pop    ebp
    ret    0
_insert    ENDP
_TEXT      ENDS
```

Nothing special. For index calculation, three input arguments are multiplying by formula $address = 600 \cdot 4 \cdot x + 30 \cdot 4 \cdot y + 4z$ to represent array as multidimensional. Do not forget that *int* type is 32-bit (4 bytes), so all coefficients should be multiplied by 4.

Listing 1.56: GCC 4.4.1

```
                public insert
insert          proc near

x               = dword ptr  8
y               = dword ptr  0Ch
z               = dword ptr  10h
value           = dword ptr  14h

                push    ebp
                mov     ebp, esp
                push    ebx
                mov     ebx, [ebp+x]
                mov     eax, [ebp+y]
                mov     ecx, [ebp+z]
                lea     edx, [eax+eax]              ; edx=y*2
                mov     eax, edx                    ; eax=y*2
                shl     eax, 4                      ; eax=(y*2)<<4 = y*2*16 = y*32
                sub     eax, edx                    ; eax=y*32 - y*2=y*30
                imul    edx, ebx, 600               ; edx=x*600
                add     eax, edx                    ; eax=eax+edx=y*30 + x*600
                lea     edx, [eax+ecx]              ; edx=y*30 + x*600 + z
                mov     eax, [ebp+value]
                mov     dword ptr ds:a[edx*4], eax  ; *(a+edx*4)=value
                pop     ebx
                pop     ebp
                retn
insert          endp
```

GCC compiler does it differently. For one of operations calculating $(30y)$, GCC produced a code without multiplication instruction. This is how it done: $(y+y) \ll 4 - (y+y) = (2y) \ll 4 - 2y = 2 \cdot 16 \cdot y - 2y = 32y - 2y = 30y$. Thus, for $30y$ calculation, only one addition operation used, one bitwise shift operation and one subtraction operation. That works faster.

ARM + Non-optimizing Xcode (LLVM) + thumb mode

Listing 1.57: Non-optimizing Xcode (LLVM) + thumb mode

```
_insert

value           = -0x10
z               = -0xC
y               = -8
x               = -4

                SUB             SP, SP, #0x10 ; allocate place in local stack for 4 int values
                MOV             R9, 0xFC2 ; a
                ADD             R9, PC
                LDR.W           R9, [R9]
                STR             R0, [SP,#0x10+x]
                STR             R1, [SP,#0x10+y]
                STR             R2, [SP,#0x10+z]
                STR             R3, [SP,#0x10+value]
                LDR             R0, [SP,#0x10+value]
                LDR             R1, [SP,#0x10+z]
                LDR             R2, [SP,#0x10+y]
                LDR             R3, [SP,#0x10+x]
                MOV             R12, 2400
                MUL.W           R3, R3, R12
                ADD             R3, R9
                MOV             R9, 120
                MUL.W           R2, R2, R9
                ADD             R2, R3
                LSLS            R1, R1, #2 ; R1=R1<<2
                ADD             R1, R2
                STR             R0, [R1]   ; R1 - address of array element
                ADD             SP, SP, #0x10 ; deallocate place in local stack for 4 int values
                BX              LR
```

Non-optimizing LLVM saves all variables in local stack, however, it's redundant. Address of array element is calculated by formula we already figured out.

ARM + Optimizing Xcode (LLVM) + thumb mode

Listing 1.58: Optimizing Xcode (LLVM) + thumb mode

```
_insert
MOVW            R9, #0x10FC
MOV.W           R12, #2400
MOVT.W          R9, #0
RSB.W           R1, R1, R1,LSL#4  ; R1 - y. R1=y<<4 - y = y*16 - y = y*15
ADD             R9, PC            ; R9 = pointer to a array
LDR.W           R9, [R9]
MLA.W           R0, R0, R12, R9   ; R0 - x, R12 - 2400, R9 - pointer to a. R0=x*2400 + ptr to a
ADD.W           R0, R0, R1,LSL#3  ; R0 = R0+R1<<3 = R0+R1*8 = x*2400 + ptr to a + y*15*8 =
                                  ; ptr to a + y*30*4 + x*600*4
STR.W           R3, [R0,R2,LSL#2] ; R2 - z, R3 - value. address=R0+z*4 =
                                  ; ptr to a + y*30*4 + x*600*4 + z*4
BX              LR
```

Here is used tricks for replacing multiplication by shift, addition and subtraction we already considered.

Here we also see new instruction for us: RSB (*Reverse Subtract*). It works just as SUB, but swapping operands with each other. Why? SUB, RSB, are those instructions, to the second operand of which shift coefficient may be applied: (LSL#4). But this coefficient may be applied only to second operand. That's fine for commutative operations like addition or multiplication, operands may be swapped there without result affecting. But subtraction is non-commutative operation, so, for these cases, RSB exist.

''LDR.W R9, [R9]'' works like LEA 2.1 in x86, but here it does nothing, it's redundant. Apparently, compiler not optimized it.

1.15 Bit fields

A lot of functions defining input flags in arguments using bit fields. Of course, it could be substituted by *bool*-typed variables set, but it's not frugally.

1.15.1 Specific bit checking

x86

Win32 API example:

```
        HANDLE fh;

        fh=CreateFile ("file", GENERIC_WRITE | GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_ALWAYS,
    FILE_ATTRIBUTE_NORMAL, NULL);
```

We got (MSVC 2010):

Listing 1.59: 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
```

Let's take a look into WinNT.h:

Listing 1.60: WinNT.h

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

Everything is clear, `GENERIC_READ | GENERIC_WRITE = 0x80000000 | 0x40000000 = 0xC0000000`, and that's value is used as second argument for `CreateFile()`[80] function.

How `CreateFile()` will check flags?

Let's take a look into KERNEL32.DLL in Windows XP SP3 x86 and we'll find this fragment of code in the function `CreateFileW`:

Listing 1.61: KERNEL32.DLL (Windows XP SP3 x86)

```
.text:7C83D429                test    byte ptr [ebp+dwDesiredAccess+3], 40h
.text:7C83D42D                mov     [ebp+var_8], 1
.text:7C83D434                jz      short loc_7C83D417
.text:7C83D436                jmp     loc_7C810817
```

Here we see `TEST` instruction, it takes, however, not the whole second argument, but only most significant byte (`ebp+dwDesiredAccess+3`) and checks it for 0x40 flag (meaning `GENERIC_WRITE` flag here)

`TEST` is merely the same instruction as `AND`, but without result saving (recall the fact `CMP` instruction is merely the same as `SUB`, but without result saving 1.4.5).

This fragment of code logic is as follows:

```
if ((dwDesiredAccess&0x40000000) == 0) goto loc_7C83D417
```

If `AND` instruction leaving this bit, `ZF` flag will be cleared and `JZ` conditional jump will not be triggered. Conditional jump will be triggered only if $0x40000000$ bit is absent in `dwDesiredAccess` variable — then `AND` result will be 0, `ZF` flag will be set and conditional jump is to be triggered.

Let's try GCC 4.4.1 and Linux:

[80] MSDN: CreateFile function

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

void main()
{
        int handle;

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

We got:

```
                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
```

[caption=GCC 4.4.1]

Let's take a look into `open()` function in `libc.so.6` library, but there is only syscall calling:

Listing 1.62: open() (libc.so.6)

```
.text:000BE69B                mov     edx, [esp+4+mode] ; mode
.text:000BE69F                mov     ecx, [esp+4+flags] ; flags
.text:000BE6A3                mov     ebx, [esp+4+filename] ; filename
.text:000BE6A7                mov     eax, 5
.text:000BE6AC                int     80h             ; LINUX - sys_open
```

So, `open()` bit fields apparently checked somewhere in Linux kernel.

Of course, it is easily to download both Glibc and Linux kernel source code, but we are interesting to understand the matter without it.

So, as of Linux 2.6, when `sys_open` syscall is called, control eventually passed into `do_sys_open` kernel function. From there — to `do_filp_open()` function (this function located in kernel source tree in the file `fs/namei.c`).

Important note. Aside from usual passing arguments via stack, there are also method to pass some of them via registers. This is also called fastcall 2.5.3. This works faster, because CPU not needed to access a stack in memory to read argument values. GCC has option *regparm*[81], and it's possible to set a number of arguments which might be passed via registers.

Linux 2.6 kernel compiled with `-mregparm=3` option [82] [83].

What it means to us, the first 3 arguments will be passed via `EAX`, `EDX` and `ECX` registers, the other ones via stack. Of course, if arguments number is less than 3, only part of registers will be used.

So, let's download Linux Kernel 2.6.31, compile it in Ubuntu: `make vmlinux`, open it in IDA 5, find the `do_filp_open()` function. At the beginning, we will see (comments are mine):

```
do_filp_open    proc near
...
                push    ebp
```

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

[82] http://kernelnewbies.org/Linux_2_6_20#head-042c62f290834eb1fe0a1942bbf5bb9a4accbc8f

[83] See also `arch\x86\include\asm\calling.h` file in kernel tree

```
                mov     ebp, esp
                push    edi
                push    esi
                push    ebx
                mov     ebx, ecx
                add     ebx, 1
                sub     esp, 98h
                mov     esi, [ebp+arg_4] ; acc_mode (5th arg)
                test    bl, 3
                mov     [ebp+var_80], eax ; dfd (1th arg)
                mov     [ebp+var_7C], edx ; pathname (2th arg)
                mov     [ebp+var_78], ecx ; open_flag (3th arg)
                jnz     short loc_C01EF684
                mov     ebx, ecx        ; ebx <- open_flag
```

[caption=do_filp_open() (linux kernel 2.6.31)]

GCC saves first 3 arguments values in local stack. Otherwise, if compiler would not touch these registers, it would be too tight environment for compiler's register allocator.

Let's find this fragment of code:

```
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
```

[caption=do_filp_open() (linux kernel 2.6.31)]

$0x40$ — is what `O_CREAT` macro equals to. `open_flag` checked for $0x40$ bit presence, and if this bit is 1, next `JNZ` instruction is triggered.

ARM

`O_CREAT` bit is checked differently in Linux kernel 3.8.0.

Listing 1.63: linux kernel 3.8.0

```
struct file *do_filp_open(int dfd, struct filename *pathname,
                const struct open_flags *op)
{
...
        filp = path_openat(dfd, pathname, &nd, op, flags | LOOKUP_RCU);
...
}

static struct file *path_openat(int dfd, struct filename *pathname,
                struct nameidata *nd, const struct open_flags *op, int flags)
{
...
        error = do_last(nd, &path, file, op, &opened, pathname);
...
}


static int do_last(struct nameidata *nd, struct path *path,
                   struct file *file, const struct open_flags *op,
                   int *opened, struct filename *name)
{
...
        if (!(open_flag & O_CREAT)) {
    ...
```

```
                error = lookup_fast(nd, path, &inode);
    ...
        } else {
    ...
                error = complete_walk(nd);
        }
...
}
```

Here is how kernel compiled for ARM mode looks like in IDA 5:

Listing 1.64: do_last() (vmlinux)

```
...
.text:C0169EA8                 MOV             R9, R3  ; R3 - (4th argument) open_flag
...
.text:C0169ED4                 LDR             R6, [R9] ; R6 - open_flag
...
.text:C0169F68                 TST             R6, #0x40 ; jumptable C0169F00 default case
.text:C0169F6C                 BNE             loc_C016A128
.text:C0169F70                 LDR             R2, [R4,#0x10]
.text:C0169F74                 ADD             R12, R4, #8
.text:C0169F78                 LDR             R3, [R4,#0xC]
.text:C0169F7C                 MOV             R0, R4
.text:C0169F80                 STR             R12, [R11,#var_50]
.text:C0169F84                 LDRB            R3, [R2,R3]
.text:C0169F88                 MOV             R2, R8
.text:C0169F8C                 CMP             R3, #0
.text:C0169F90                 ORRNE           R1, R1, #3
.text:C0169F94                 STRNE           R1, [R4,#0x24]
.text:C0169F98                 ANDS            R3, R6, #0x200000
.text:C0169F9C                 MOV             R1, R12
.text:C0169FA0                 LDRNE           R3, [R4,#0x24]
.text:C0169FA4                 ANDNE           R3, R3, #1
.text:C0169FA8                 EORNE           R3, R3, #1
.text:C0169FAC                 STR             R3, [R11,#var_54]
.text:C0169FB0                 SUB             R3, R11, #-var_38
.text:C0169FB4                 BL              lookup_fast
...
.text:C016A128 loc_C016A128                            ; CODE XREF: do_last.isra.14+DC
.text:C016A128                 MOV             R0, R4
.text:C016A12C                 BL              complete_walk
...
```

TST is analogical to TEST instruction in x86.

We can "spot" visually this code fragment by the fact that `lookup_fast()` will be executed in one case and `complete_walk()` in another case. This is corresponding to `do_last()` function source code.

O_CREAT macro is equals to $0x40$ here too.

1.15.2 Specific bit setting/clearing

For example:

```
#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;
};
```

x86

We got (MSVC 2010):

Listing 1.65: 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
```

OR instruction adding one more bit to value, ignoring others.

AND resetting one bit. It can be said, AND just copies all bits except one. Indeed, in the second AND operand only those bits are set, which are needed to be saved, except one bit we wouldn't like to copy (which is 0 in bitmask). It's easier way to memorize the logic.

If we compile it in MSVC with optimization turned on (/Ox), the code will be even shorter:

Listing 1.66: Optimizing MSVC

```
_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
```

Let's try GCC 4.4.1 without optimization:

Listing 1.67: Non-optimizing GCC

```
                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
```

There are some redundant code present, however, it's shorter then MSVC version without optimization.

Now let's try GCC with optimization turned on -O3:

Listing 1.68: Optimizing GCC

```
                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
```

That's shorter. It is important to note that compiler works with EAX register part via AH register — that's EAX register part from 8th to 15th bits inclusive.

Important note: 16-bit CPU 8086 accumulator was named AX and consisted of two 8-bit halves — AL (lower byte) and AH (higher byte). In 80386 almost all regsiters were extended to 32-bit, accumulator was named EAX, but for the sake of compatibility, its *older parts* may be still accessed as AX/AH/AL registers.

Because all x86 CPUs are 16-bit 8086 CPU successors, these *older* 16-bit opcodes are shorter than newer 32-bit opcodes. That's why ''or ah, 40h'' instruction occupying only 3 bytes. It would be more logical way to emit here ''or eax, 04000h'', but that's 5 bytes, or even 6 (if register in first operand is not EAX).

It would be even shorter if to turn on -O3 optimization flag and also set regparm=3.

Listing 1.69: Optimizing GCC

```
                public f
f               proc near
                push    ebp
                or      ah, 40h
                mov     ebp, esp
                and     ah, 0FDh
                pop     ebp
                retn
f               endp
```

Indeed — first argument is already loaded into EAX, so it's possible to work with it in-place. It's worth noting that both function prologue (''push ebp / mov ebp,esp'') and epilogue (''pop ebp'') can easily be omitted here, but GCC probably isn't good enough for such code size optimizations. However, such short functions are better to be *inlined functions*[84].

ARM + Optimizing Keil + ARM mode

Listing 1.70: Optimizing Keil + ARM mode

```
02 0C C0 E3          BIC     R0, R0, #0x200
01 09 80 E3          ORR     R0, R0, #0x4000
1E FF 2F E1          BX      LR
```

BIC is "logical and", analogical to AND in x86. ORR is "logical or", analogical to OR in x86.

So far, so easy.

ARM + Optimizing Keil + thumb mode

Listing 1.71: Optimizing Keil + thumb mode

```
01 21 89 03          MOVS    R1, 0x4000
08 43                ORRS    R0, R1
49 11                ASRS    R1, R1, #5   ; generate 0x200 and place to R1
```

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

```
88 43               BICS    R0, R1
70 47               BX      LR
```

Apparently, Keil concludes that the code in thumb mode, making $0x200$ from $0x4000$, will be more compact than code, writting $0x200$ to arbitrary register.

So that's why, with the help of ASRS (arithmetic shift right), this value is calculating as $0x4000 \gg 5$.

ARM + Optimizing Xcode (LLVM) + ARM mode

Listing 1.72: Optimizing Xcode (LLVM) + ARM mode

```
42 0C C0 E3         BIC             R0, R0, #0x4200
01 09 80 E3         ORR             R0, R0, #0x4000
1E FF 2F E1         BX              LR
```

The code was generated by LLVM, in source code form, in fact, could be looks like:

```
    REMOVE_BIT (rt, 0x4200);
    SET_BIT (rt, 0x4000);
```

And it does exactly the same we need. But why $0x4200$? Perhaps, that's LLVM optimizer's artifact [85]. Probably, compiler's optimizer error, but generated code works correct anyway.

More about compiler's anomalies, read here 8.

For thumb mode, Optimizing Xcode (LLVM) generates exactly the same code.

1.15.3 Shifts

Bit shifts in C/C++ are implemented via $\ll$ and $\gg$ operators.

Here is a simple example of function, calculating number of 1 bits in input variable:

```
#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;
};
```

In this loop, iteration count value *i* counting from 0 to 31, $1 \ll i$ statement will be counting from 1 to $0x80000000$. Describing this operation in natural language, we would say *shift* 1 *by n bits left*. In other words, $1 \ll i$ statement will consequently produce all possible bit positions in 32-bit number. By the way, freed bit at right is always cleared. IS_SET macro is checking bit presence in a.

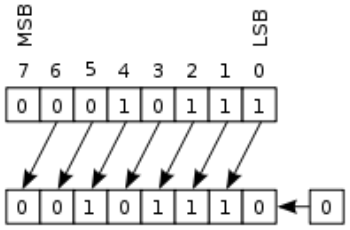

Figure 1.1: How SHL instruction works[86]

[85] It was LLVM build 2410.2.00 bundled with Apple Xcode 4.6.3

[86] illustration taken from wikipedia

The IS_SET macro is in fact logical and operation (*AND*) and it returns zero if specific bit is absent there, or bit mask, if the bit is present. *if()* operator triggered in C/C++ if expression in it isn't zero, it might be even 123456, that's why it always working correctly.

x86

Let's compile (MSVC 2010):

Listing 1.73: MSVC 2010

```
_rt$ = -8               ; size = 4
_i$ = -4                ; size = 4
_a$ = 8                 ; size = 4
_f  PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 8
    mov    DWORD PTR _rt$[ebp], 0
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN4@f
$LN3@f:
    mov    eax, DWORD PTR _i$[ebp]   ; 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
```

That's how SHL (*SHift Left*) working.

Let's compile it in GCC 4.4.1:

Listing 1.74: 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
```

Shift instructions are often used in division and multiplications by power of two numbers (1, 2, 4, 8, etc).

For example:

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

We got (MSVC 2010):

Listing 1.75: MSVC 2010

```
_a$ = 8                                                 ; size = 4
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        shr     eax, 2
        ret     0
_f      ENDP
```

SHR (*SHift Right*) instruction in this example is shifting a number by 2 bits right. Two freed bits at left (e.g., two most significant bits) are set to zero. Two least significant bits are dropped. In fact, these two dropped bits — division operation remainder.

SHR instruction works just like as SHL but in other direction.

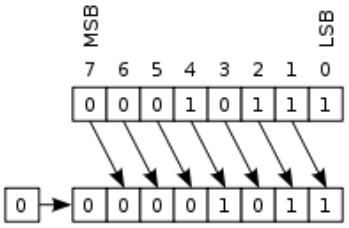

Figure 1.2: How SHR instruction works[87]

It can be easily understood if to imagine decimal numeral system and number 23. 23 can be easily divided by 10 just by dropping last digit (3 — is division remainder). 2 is leaving after operation as a quotient [88].

The same story about multiplication. Multiplication by 4 is just shifting the number to the left by 2 bits, inserting 2 zero bits at right (as the last two bits). It's just like to multiply 3 by 100 — we need just to add two zeroes at the right.

[87] illustration taken from wikipedia

[88] division result

ARM + Optimizing Xcode (LLVM) + ARM mode

Listing 1.76: Optimizing Xcode (LLVM) + ARM mode

```
                MOV             R1, R0
                MOV             R0, #0
                MOV             R2, #1
                MOV             R3, R0
loc_2E54
                TST             R1, R2,LSL R3 ; set flags according to R1 & (R2<<R3)
                ADD             R3, R3, #1    ; R3++
                ADDNE           R0, R0, #1    ; if ZF flag is cleared by TST, R0++
                CMP             R3, #32
                BNE             loc_2E54
                BX              LR
```

TST is the same things as TEST in x86.

As I mentioned before 1.12.1, there are no separate shifting instructions in ARM mode. However, there are modifiers LSL (*Logical Shift Left*), LSR (*Logical Shift Right*), ASR (*Arithmetic Shift Right*), ROR (*Rotate Right*) and RRX (*Rotate Right with Extend*) , which may be added to such instructions as MOV, TST, CMP, ADD, SUB, RSB[89].

These modificators are defines, how to shift second operand and by how many bits.

Thus "TST R1, R2,LSL R3" instruction works here as $R1 \wedge (R2 \ll R3)$.

ARM + Optimizing Xcode (LLVM) + thumb-2 mode

Almost the same, but here are two LSL.W/TST instructions are used instead of single TST, becase, in thumb mode, it's not possible to define LSL modifier right in TST.

```
                MOV             R1, R0
                MOVS            R0, #0
                MOV.W           R9, #1
                MOVS            R3, #0
loc_2F7A
                LSL.W           R2, R9, R3
                TST             R2, R1
                ADD.W           R3, R3, #1
                IT NE
                ADDNE           R0, #1
                CMP             R3, #32
                BNE             loc_2F7A
                BX              LR
```

1.15.4 CRC32 calculation example

This is very popular table-based CRC32 hash calculation method[90].

```
/* 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,
```

[89] These instructions are also called "data processing instructions"

[90] Source code was taken here: http://burtleburtle.net/bob/c/crc.c

```
  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,
  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;
}
```

We are interesting in `crc()` function only. By the way, pay attention to two loop initializers in `for()` statement: `hash=len, i=0`. C/C++ standard allows this, of course. Emited code will contain two operations in loop initialization part instead of usual one.

Let's compile it in MSVC with optimization (`/Ox`). For the sake of brevity, only `crc()` function is listed here, with my comments.

```
_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

; EAX=EAX>>8; bits 24-31 taken "from nowhere" will be cleared
    shr    eax, 8

; EAX=EAX^crctab[EDI*4] - choose EDI-th element from crctab[] table
    xor    eax, DWORD PTR _crctab[edi*4]
    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
```

Let's try the same in GCC 4.4.1 with `-O3` option:

```
                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 aligned loop start by 8-byte border by adding `NOP` and `lea esi, [esi+0]` (that's *idle operation* too). Read more about it in npad section 2.3.

1.16 Structures

It can be defined that C/C++ structure, with some assumptions, just a set of variables, always stored in memory together, not necessary of the same type.

1.16.1 SYSTEMTIME example

Let's take SYSTEMTIME[91] win32 structure describing time.

That's how it's defined:

Listing 1.77: WinBase.h

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Let's write a C function to get current time:

```
#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;
};
```

We got (MSVC 2010):

Listing 1.78: 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
```

[91] MSDN: SYSTEMTIME structure

```
    push   OFFSET $SG78811 ; '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
    call   _printf
    add    esp, 28      ; 0000001cH
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main      ENDP
```

16 bytes are allocated for this structure in local stack — that's exactly `sizeof(WORD)*8` (there are 8 WORD variables in the structure).

Pay attention to the fact the structure beginning with `wYear` field. It can be said, an pointer to SYSTEMTIME structure is passed to `GetSystemTime()`[92], but it's also can be said, pointer to `wYear` field is passed, and that's the same! `GetSystemTime()` writting current year to the WORD pointer pointing to, then shifting 2 bytes ahead, then writting current month, etc, etc.

The fact that structure fields are just variables located side-by-side, I can illustrate by the following method. Keeping in ming `SYSTEMTIME` structure description, I can rewrite this simple example like this:

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

void main()
{
    WORD array[8];
    GetSystemTime (array);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        array[0] /* wYear */, array[1] /* wMonth */, array[3] /* wDay */,
        array[4] /* wHour */, array[5] /* wMinute */, array[6] /* wSecond */);

    return;
};
```

Compiler will grumble for a little:

```
systemtime2.c(7) : warning C4133: 'function' : incompatible types - from 'WORD [8]' to 'LPSYSTEMTIME'
```

But nevertheless, it will produce this code:

Listing 1.79: MSVC 2010

```
$SG78573 DB     '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H

_array$ = -16                                           ; size = 16
_main   PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 16                                 ; 00000010H
        lea     eax, DWORD PTR _array$[ebp]
        push    eax
        call    DWORD PTR __imp__GetSystemTime@4
        movzx   ecx, WORD PTR _array$[ebp+12] ; wSecond
        push    ecx
        movzx   edx, WORD PTR _array$[ebp+10] ; wMinute
        push    edx
        movzx   eax, WORD PTR _array$[ebp+8] ; wHoure
        push    eax
        movzx   ecx, WORD PTR _array$[ebp+6] ; wDay
        push    ecx
        movzx   edx, WORD PTR _array$[ebp+2] ; wMonth
        push    edx
        movzx   eax, WORD PTR _array$[ebp] ; wYear
        push    eax
        push    OFFSET $SG78573
```

[92] MSDN: SYSTEMTIME structure

```
        call    _printf
        add     esp, 28                                 ; 0000001cH
        xor     eax, eax
        mov     esp, ebp
        pop     ebp
        ret     0
_main   ENDP
```

And it works just as the same!

It's very interesting fact that result in assembly form cannot be distinguished from the result of previous compilation. So by looking at this code, one cannot say for sure, was there structure declared, or just pack of variables.

Nevertheless, no one will do it in sane state of mind. Because it's not convenient. Also, structure fields may be changed by developers, swapped, etc.

1.16.2 Let's allocate place for structure using malloc()

However, sometimes it's simpler to place structures not in local stack, but in heap:

```
#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;
};
```

Let's compile it now with optimization (`/Ox`) so to easily see what we need.

Listing 1.80: Optimizing MSVC

```
_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
```

```
    call   _free
    add    esp, 32                  ; 00000020H
    xor    eax, eax
    pop    esi
    ret    0
_main      ENDP
```

So, `sizeof(SYSTEMTIME) = 16`, that's exact number of bytes to be allocated by `malloc()`. It return the pointer to freshly allocated memory block in `EAX`, which is then moved into `ESI`. `GetSystemTime()` win32 function undertake to save `ESI` value, and that's why it is not saved here and continue to be used after `GetSystemTime()` call.

New instruction — `MOVZX` (*Move with Zero eXtent*). It may be used almost in those cases as `MOVSX` 1.11.1, but, it clearing other bits to 0. That's because `printf()` require 32-bit *int*, but we got WORD in structure — that's 16-bit unsigned type. That's why by copying value from WORD into *int*, bits from 16 to 31 should be cleared, because there will be random noise otherwise, leaved from previous operations on registers.

In this example, I can represent structure as array of WORD-s:

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

void main()
{
    WORD *t;

    t=(WORD *)malloc (16);

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t[0] /* wYear */, t[1] /* wMonth */, t[3] /* wDay */,
        t[4] /* wHour */, t[5] /* wMinute */, t[6] /* wSecond */);

    free (t);

    return;
};
```

We got:

Listing 1.81: Optimizing MSVC

```
$SG78594 DB     '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H

_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]
        movzx   ecx, WORD PTR [esi+10]
        movzx   edx, WORD PTR [esi+8]
        push    eax
        movzx   eax, WORD PTR [esi+6]
        push    ecx
        movzx   ecx, WORD PTR [esi+2]
        push    edx
        movzx   edx, WORD PTR [esi]
        push    eax
        push    ecx
        push    edx
        push    OFFSET $SG78594
        call    _printf
        push    esi
        call    _free
```

```
        add     esp, 32                                 ; 00000020H
        xor     eax, eax
        pop     esi
        ret     0
_main   ENDP
```

Again, we got a code that cannot be distinguished from previous. And again I should note, one shouldn't do this in practice.

1.16.3 struct tm

Linux

As of Linux, let's take `tm` structure from `time.h` for example:

```
#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);
};
```

Let's compile it in GCC 4.4.1:

```
main            proc near
                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 40h
                mov     dword ptr [esp], 0 ; first argument for time()
                call    time
                mov     [esp+3Ch], eax
                lea     eax, [esp+3Ch]  ; take pointer to what time() returned
                lea     edx, [esp+10h]  ; at ESP+10h struct tm will begin
                mov     [esp+4], edx    ; pass pointer to the structure begin
                mov     [esp], eax      ; pass pointer to result of 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
```

Somehow, IDA 5 didn't created local variables names in local stack. But since we already experienced reverse engineers :-) we may do it without this information in this simple example.

Please also pay attention to `lea edx, [eax+76Ch]` — this instruction just adding $0x76C$ to `EAX`, but not modify any flags. See also relevant section about `LEA` 2.1.

In order to illustrate that structure is just variables laying side-by-side in one place, let's rework example, while looking at the file *time.h*:

Listing 1.82: time.h

```
struct tm
{
  int   tm_sec;
  int   tm_min;
  int   tm_hour;
  int   tm_mday;
  int   tm_mon;
  int   tm_year;
  int   tm_wday;
  int   tm_yday;
  int   tm_isdst;
};
```

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

void main()
{
    int tm_sec, tm_min, tm_hour, tm_mday, tm_mon, tm_year, tm_wday, tm_yday, tm_isdst;
    time_t unix_time;

    unix_time=time(NULL);

    localtime_r (&unix_time, &tm_sec);

    printf ("Year: %d\n", tm_year+1900);
    printf ("Month: %d\n", tm_mon);
    printf ("Day: %d\n", tm_mday);
    printf ("Hour: %d\n", tm_hour);
    printf ("Minutes: %d\n", tm_min);
    printf ("Seconds: %d\n", tm_sec);
};
```

Please note that pointer to exactly `tm_sec` is passed into `localtime_r`, i.e., to the first "structure" element.

Compiler will warn us:

Listing 1.83: GCC 4.7.3

```
GCC_tm2.c: In function 'main':
GCC_tm2.c:11:5: warning: passing argument 2 of 'localtime_r' from incompatible pointer type [enabled by
    default]
In file included from GCC_tm2.c:2:0:
/usr/include/time.h:59:12: note: expected 'struct tm *' but argument is of type 'int *'
```

But nevertheless, will generate this:

Listing 1.84: GCC 4.7.3

```
main            proc near

var_30          = dword ptr -30h
var_2C          = dword ptr -2Ch
unix_time       = dword ptr -1Ch
tm_sec          = dword ptr -18h
tm_min          = dword ptr -14h
tm_hour         = dword ptr -10h
tm_mday         = dword ptr -0Ch
tm_mon          = dword ptr -8
tm_year         = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 30h
                call    __main
                mov     [esp+30h+var_30], 0 ; arg 0
                call    time
                mov     [esp+30h+unix_time], eax
                lea     eax, [esp+30h+tm_sec]
                mov     [esp+30h+var_2C], eax
                lea     eax, [esp+30h+unix_time]
                mov     [esp+30h+var_30], eax
                call    localtime_r
                mov     eax, [esp+30h+tm_year]
                add     eax, 1900
                mov     [esp+30h+var_2C], eax
                mov     [esp+30h+var_30], offset aYearD ; "Year: %d\n"
                call    printf
                mov     eax, [esp+30h+tm_mon]
                mov     [esp+30h+var_2C], eax
                mov     [esp+30h+var_30], offset aMonthD ; "Month: %d\n"
                call    printf
                mov     eax, [esp+30h+tm_mday]
                mov     [esp+30h+var_2C], eax
                mov     [esp+30h+var_30], offset aDayD ; "Day: %d\n"
                call    printf
                mov     eax, [esp+30h+tm_hour]
                mov     [esp+30h+var_2C], eax
                mov     [esp+30h+var_30], offset aHourD ; "Hour: %d\n"
                call    printf
                mov     eax, [esp+30h+tm_min]
                mov     [esp+30h+var_2C], eax
                mov     [esp+30h+var_30], offset aMinutesD ; "Minutes: %d\n"
                call    printf
                mov     eax, [esp+30h+tm_sec]
                mov     [esp+30h+var_2C], eax
                mov     [esp+30h+var_30], offset aSecondsD ; "Seconds: %d\n"
                call    printf
                leave
                retn
main            endp
```

This code is identical to what we saw previously and it's not possible to say, was it structure in original source code or just pack of variables.

And this works. However, it's not recommended to do this in practice. Usually, compiler allocate variables in local stack in the same order as they were declared in function. Nevertheless, there are no any guarantee.

By the way, some other compiler may warn that `tm_year`, `tm_mon`, `tm_mday`, `tm_hour`, `tm_min` variables, but not `tm_sec` are used without being initialized. Indeed, compiler don't know that these will be filled when calling to `localtime_r()`.

I chose exactly this example for illustration, because all structure fields has *int* type, and `SYSTEMTIME` structure fields — 16-bit `WORD`, and if to declare them as a local variables, they will be aligned by 32-bit border, and nothing will work (because `GetSystemTime()` will fill them incorrectly). Read more about it in next section: "Fields packing in structure".

So, structure is just variables pack laying on one place, side-by-side. I could say that structure is a syntactic sugar, directing compiler to hold them in one place. However, I'm not programming languages expert, so, most likely, I'm wrong with this term. By the way, there were a times, in very early C versions (before 1972), in which there were no structures at all [Rit93].

ARM + Optimizing Keil + thumb mode

Same example:

Listing 1.85: Optimizing Keil + thumb mode

```
var_38          = -0x38
var_34          = -0x34
var_30          = -0x30
var_2C          = -0x2C
var_28          = -0x28
var_24          = -0x24
timer           = -0xC

                PUSH    {LR}
                MOVS    R0, #0          ; timer
                SUB     SP, SP, #0x34
                BL      time
                STR     R0, [SP,#0x38+timer]
                MOV     R1, SP          ; tp
                ADD     R0, SP, #0x38+timer ; timer
                BL      localtime_r
                LDR     R1, =0x76C
                LDR     R0, [SP,#0x38+var_24]
                ADDS    R1, R0, R1
                ADR     R0, aYearD      ; "Year: %d\n"
                BL      __2printf
                LDR     R1, [SP,#0x38+var_28]
                ADR     R0, aMonthD     ; "Month: %d\n"
                BL      __2printf
                LDR     R1, [SP,#0x38+var_2C]
                ADR     R0, aDayD       ; "Day: %d\n"
                BL      __2printf
                LDR     R1, [SP,#0x38+var_30]
                ADR     R0, aHourD      ; "Hour: %d\n"
                BL      __2printf
                LDR     R1, [SP,#0x38+var_34]
                ADR     R0, aMinutesD   ; "Minutes: %d\n"
                BL      __2printf
                LDR     R1, [SP,#0x38+var_38]
                ADR     R0, aSecondsD   ; "Seconds: %d\n"
                BL      __2printf
                ADD     SP, SP, #0x34
                POP     {PC}
```

ARM + Optimizing Xcode (LLVM) + thumb-2 mode

IDA 5 "get to know" `tm` structure (because IDA 5 "knows" argument types of library functions like `localtime_r()`), so it shows here structure elements accesses and also names are assigned to them.

Listing 1.86: Optimizing Xcode (LLVM) + thumb-2 mode

```
var_38          = -0x38
var_34          = -0x34

                PUSH            {R7,LR}
                MOV             R7, SP
                SUB             SP, SP, #0x30
                MOVS            R0, #0  ; time_t *
                BLX             _time
                ADD             R1, SP, #0x38+var_34 ; struct tm *
                STR             R0, [SP,#0x38+var_38]
                MOV             R0, SP  ; time_t *
                BLX             _localtime_r
                LDR             R1, [SP,#0x38+var_34.tm_year]
                MOV             R0, 0xF44 ; "Year: %d\n"
                ADD             R0, PC  ; char *
                ADDW            R1, R1, #0x76C
                BLX             _printf
                LDR             R1, [SP,#0x38+var_34.tm_mon]
                MOV             R0, 0xF3A ; "Month: %d\n"
                ADD             R0, PC  ; char *
                BLX             _printf
                LDR             R1, [SP,#0x38+var_34.tm_mday]
                MOV             R0, 0xF35 ; "Day: %d\n"
                ADD             R0, PC  ; char *
                BLX             _printf
                LDR             R1, [SP,#0x38+var_34.tm_hour]
                MOV             R0, 0xF2E ; "Hour: %d\n"
                ADD             R0, PC  ; char *
                BLX             _printf
                LDR             R1, [SP,#0x38+var_34.tm_min]
                MOV             R0, 0xF28 ; "Minutes: %d\n"
                ADD             R0, PC  ; char *
                BLX             _printf
                LDR             R1, [SP,#0x38+var_34]
                MOV             R0, 0xF25 ; "Seconds: %d\n"
                ADD             R0, PC  ; char *
                BLX             _printf
                ADD             SP, SP, #0x30
                POP             {R7,PC}

...

00000000 tm              struc ; (sizeof=0x2C, standard type)
00000000 tm_sec          DCD ?
00000004 tm_min          DCD ?
00000008 tm_hour         DCD ?
0000000C tm_mday         DCD ?
00000010 tm_mon          DCD ?
00000014 tm_year         DCD ?
00000018 tm_wday         DCD ?
0000001C tm_yday         DCD ?
00000020 tm_isdst        DCD ?
00000024 tm_gmtoff       DCD ?
00000028 tm_zone         DCD ?                   ; offset
0000002C tm              ends
```

1.16.4 Fields packing in structure

One important thing is fields packing in structures[93].

Let's take a simple example:

```
#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);
};
```

As we see, we have two *char* fields (each is exactly one byte) and two more — *int* (each - 4 bytes).

x86

That's all compiling into:

```
_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
    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
```

As we can see, each field's address is aligned by 4-bytes border. That's why each *char* using 4 bytes here, like *int*. Why? Thus it's easier for CPU to access memory at aligned addresses and to cache data from it.

However, it's not very economical in size sense.

Let's try to compile it with option (`/Zp1`) (*/Zp[n] pack structs on n-byte boundary*).

Listing 1.87: MSVC /Zp1

```
_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
```

[93] See also: Wikipedia: Data structure alignment

```
    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
```

Now the structure takes only 10 bytes and each *char* value takes 1 byte. What it give to us? Size economy. And as drawback — CPU will access these fields without maximal performance it can.

As it can be easily guessed, if the structure is used in many source and object files, all these should be compiled with the same convention about structures packing.

Aside from MSVC `/Zp` option which set how to align each structure field, here is also `#pragma pack` compiler option, it can be defined right in source code. It's available in both MSVC[94] and GCC[95].

Let's back to `SYSTEMTIME` structure consisting in 16-bit fields. How our compiler know to pack them on 1-byte alignment method?

`WinNT.h` file has this:

Listing 1.88: WinNT.h

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

And this:

Listing 1.89: WinNT.h

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

The file PshPack1.h looks like:

Listing 1.90: PshPack1.h

```
#if ! (defined(lint) || defined(RC_INVOKED))
#if ( _MSC_VER >= 800 && !defined(_M_I86)) || defined(_PUSHPOP_SUPPORTED)
#pragma warning(disable:4103)
#if !(defined( MIDL_PASS )) || defined( __midl )
#pragma pack(push,1)
#else
#pragma pack(1)
#endif
#else
#pragma pack(1)
#endif
#endif /* ! (defined(lint) || defined(RC_INVOKED)) */
```

That's how compiler will pack structures defined after `#pragma pack`.

ARM + Optimizing Keil + thumb mode

Listing 1.91: Optimizing Keil + thumb mode

```
.text:0000003E            exit                                    ; CODE XREF: f+16
.text:0000003E 05 B0                      ADD     SP, SP, #0x14
.text:00000040 00 BD                      POP     {PC}


.text:00000280            f
.text:00000280
.text:00000280            var_18          = -0x18
.text:00000280            a               = -0x14
.text:00000280            b               = -0x10
```

[94] MSDN: Working with Packing Structures
[95] Structure-Packing Pragmas

```
.text:00000280             c               = -0xC
.text:00000280             d               = -8
.text:00000280
.text:00000280 0F B5                       PUSH    {R0-R3,LR}
.text:00000282 81 B0                       SUB     SP, SP, #4
.text:00000284 04 98                       LDR     R0, [SP,#16]    ; d
.text:00000286 02 9A                       LDR     R2, [SP,#8]     ; b
.text:00000288 00 90                       STR     R0, [SP]
.text:0000028A 68 46                       MOV     R0, SP
.text:0000028C 03 7B                       LDRB    R3, [R0,#12]    ; c
.text:0000028E 01 79                       LDRB    R1, [R0,#4]     ; a
.text:00000290 59 A0                       ADR     R0, aADBDCDDD   ; "a=%d; b=%d; c=%d; d=%d\n"
.text:00000292 05 F0 AD FF                 BL      __2printf
.text:00000296 D2 E6                       B       exit
```

As we may recall, here a structure passed instead of pointer to structure, and because first 4 function arguments in ARM are passed via registers, so then structure fields are passed via R0-R3.

LDRB loads one byte from memory and extending it to 32-bit, taking into account its sign. This is similar to MOVSX 1.11.1 instruction in x86. Here it's used for loading fields a and c from structure.

One more thing we spot easily, instead of function epilogue, here is jump to another function's epilogue! Indeed, that was another function, not related in any way to our function, however, it has exactly the same epilogue (probably because, it hold 5 local variables too ($5 * 4 = 0x14$)). Also, it's located nearly (take a look on addresses). Indeed, there are no difference, which epilogue to execute, if it works just as we need. Apparently, Keil decides to do this because of economy. Epilogue takes 4 bytes while jump — only 2.

ARM + Optimizing Xcode (LLVM) + thumb-2 mode

Listing 1.92: Optimizing Xcode (LLVM) + thumb-2 mode

```
var_C           = -0xC

                PUSH            {R7,LR}
                MOV             R7, SP
                SUB             SP, SP, #4
                MOV             R9, R1 ; b
                MOV             R1, R0 ; a
                MOVW            R0, #0xF10 ; "a=%d; b=%d; c=%d; d=%d\n"
                SXTB            R1, R1  ; prepare a
                MOVT.W          R0, #0
                STR             R3, [SP,#0xC+var_C] ; place d to stack for printf()
                ADD             R0, PC  ; format-string
                SXTB            R3, R2  ; prepare c
                MOV             R2, R9  ; b
                BLX             _printf
                ADD             SP, SP, #4
                POP             {R7,PC}
```

SXTB (*Signed Extend Byte*) is analogous to MOVSX 1.11.1 in x86 as well, but works not with memory, but with register. All the rest — just the same.

1.16.5 Nested structures

Now what about situations when one structure define another structure inside?

```
#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);
};
```

...in this case, both `inner_struct` fields will be placed between a,b and d,e fields of `outer_struct`.

Let's compile (MSVC 2010):

Listing 1.93: 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
```

One curious point here is that by looking onto this assembly code, we do not even see that another structure was used inside of it! Thus, we would say, nested structures are finally unfolds into *linear* or *one-dimensional* structure.

Of course, if to replace `struct inner_struct c;` declaration to `struct inner_struct *c;` (thus making a pointer here) situation will be significally different.

1.16.6 Bit fields in structure

CPUID example

C/C++ language allow to define exact number of bits for each structure fields. It's very useful if one need to save memory space. For example, one bit is enough for variable of *bool* type. But of course, it's not rational if speed is important.

Let's consider `CPUID`[96]instruction example. This instruction return information about current CPU and its features.

If `EAX` is set to 1 before instruction execution, `CPUID` will return this information packed into `EAX` register:

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

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

MSVC 2010 has CPUID macro, but GCC 4.4.1 — hasn't. So let's make this function by yourself for GCC, using its built-in assembler[97].

```
#include <stdio.h>

#ifdef __GNUC__
static inline void cpuid(int code, int *a, int *b, int *c, int *d) {
  asm volatile("cpuid":"=a"(*a),"=b"(*b),"=c"(*c),"=d"(*d):"a"(code));
}
#endif

#ifdef _MSC_VER
#include <intrin.h>
#endif

struct CPUID_1_EAX
{
    unsigned int stepping:4;
    unsigned int model:4;
    unsigned int family_id:4;
    unsigned int processor_type:2;
    unsigned int reserved1:2;
    unsigned int extended_model_id:4;
    unsigned int extended_family_id:8;
    unsigned int reserved2:4;
};

int main()
{
    struct CPUID_1_EAX *tmp;
    int b[4];

#ifdef _MSC_VER
    __cpuid(b,1);
#endif

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

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

    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;
};
```

After CPUID will fill EAX/EBX/ECX/EDX, these registers will be reflected in b[] array. Then, we have a pointer to CPUID_1_EAX structure and we point it to EAX value from b[] array.

In other words, we treat 32-bit *int* value as a structure.

Then we read from the stucture.

[97] More about internal GCC assembler

Let's compile it in MSVC 2008 with `/Ox` option:

Listing 1.94: Optimizing MSVC 2008

```
_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
```

SHR instruction shifting value in EAX by number of bits should be *skipped*, e.g., we ignore some bits *at right*. AND instruction clearing not needed bits *at left*, or, in other words, leave only those bits in EAX we need now. Let's try GCC 4.4.1 with -O3 option.

Listing 1.95: Optimizing GCC 4.4.1

```
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
```

Almost the same. The only thing to note is that GCC somehow united calculation of `extended_model_id` and `extended_family_id` into one block, instead of calculating them separately, before corresponding each `printf()` call.

Working with the float type as with a structure

As it was already noted in section about FPU 1.13, both *float* and *double* types consisted of sign, significand (or fraction) and exponent. But will we able to work with these fields directly? Let's try with *float*.

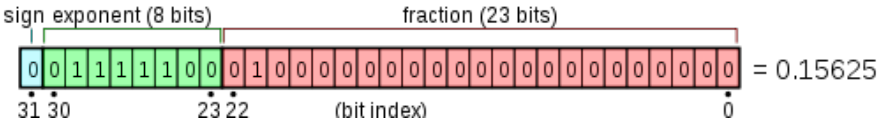

Figure 1.3: float value format[98]

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

`float_as_struct` structure occupies as much space is memory as *float*, e.g., 4 bytes or 32 bits.

Now we setting negative sign in input value and also by addding 2 to exponent we thereby multiplicating the whole number by 2^2, e.g., by 4.

Let's compile in MSVC 2008 without optimization:

Listing 1.96: Non-optimizing MSVC 2008

```
_t$ = -8                ; size = 4
_f$ = -4                ; size = 4
__in$ = 8               ; size = 4
?f@@YAMM@Z PROC         ; f
    push   ebp
```

[98] illustration taken from wikipedia

```
    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$[ebp]
    or     edx, -2147483648 ; 80000000H - set minus sign
    mov    DWORD PTR _t$[ebp], edx

    mov    eax, DWORD PTR _t$[ebp]
    shr    eax, 23          ; 00000017H - drop significand
    and    eax, 255         ; 000000ffH - leave here only exponent
    add    eax, 2           ; add 2 to it
    and    eax, 255         ; 000000ffH
    shl    eax, 23          ; 00000017H - shift result to place of bits 30:23
    mov    ecx, DWORD PTR _t$[ebp]
    and    ecx, -2139095041 ; 807fffffH - drop exponent
    or     ecx, eax         ; add original value without exponent with new calculated explonent
    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
```

Redundant for a bit. If it compiled with `/Ox` flag there are no `memcpy()` call, f variable is used directly. But it's easier to understand it all considering unoptimized version.

What GCC 4.4.1 with `-O3` will do?

Listing 1.97: Optimizing GCC 4.4.1

```
; f(float)
       public _Z1ff
_Z1ff  proc near

var_4  = dword ptr -4
arg_0  = dword ptr  8

       push    ebp
       mov     ebp, esp
       sub     esp, 4
       mov     eax, [ebp+arg_0]
       or      eax, 80000000h  ; 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
        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
```

The `f()` function is almost understandable. However, what is interesting, GCC was able to calculate `f(1.234)` result during compilation stage despite all this hodge-podge with structure fields and prepared this argument to `printf()` as precalculated!

1.17 C++ classes

1.17.1 Simple example

I placed a C++ classes description here intentionally after structures description, because internally, C++ classes representation is almost the same as structures representation.

Let's try an example with two variables, two constructors and one method:

```
#include <stdio.h>

class c
{
private:
    int v1;
    int v2;
public:
    c() // default 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;
};
```

Here is how `main()` function looks like translated into assembly language:

```
_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
```

So what's going on. For each object (instance of class *c*) 8 bytes allocated, that's exactly size of 2 variables storage.

For *c1* a default argumentless constructor `??0c@@QAE@XZ` is called. For *c2* another constructor `??0c@@QAE@HH@Z` is called and two numbers are passed as arguments.

A pointer to object (*this* in C++ terminology) is passed in `ECX` register. This is called thiscall 2.5.4 — a pointer to object passing method.

MSVC doing it using `ECX` register. Needless to say, it's not a standardized method, other compilers could do it differently, e.g., via first function argument (like GCC).

Why these functions has so odd names? That's *name mangling*[99].

C++ class may contain several methods sharing the same name but having different arguments — that's polymorphism. And of course, different classes may own methods sharing the same name.

Name mangling enable us to encode class name + method name + all method argument types in one ASCII-string, which will be used as internal function name. That's all because neither linker, nor DLL operation system loader (mangled names may be among DLL exports as well) knows nothing about C++ or OOP.

`dump()` function called two times after.

Now let's see constructors' code:

```
_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
```

Constructors are just functions, they use pointer to structure in `ECX`, moving the pointer into own local variable, however, it's not necessary.

Now `dump()` method:

[99] Wikipedia: Name mangling

```
_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
```

Simple enough: `dump()` taking pointer to the structure containing two *int*'s in ECX, takes two values from it and passing it into `printf()`.

The code is much shorter if compiled with optimization (`/Ox`):

```
??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
```

That's all. One more thing to say is that stack pointer wasn't corrected with `add esp, X` after constructor called. Withal, constructor has `ret 8` instead of RET at the end.

This is all because here used thiscall 2.5.4 calling convention, the method of passing values through the stack, which is, together with stdcall 2.5.2 method, offers to correct stack to callee rather then to caller. `ret x` instruction adding X to ESP, then passes control to caller function.

See also section about calling conventions 2.5.

It's also should be noted that compiler deciding when to call constructor and destructor — but that's we already know from C++ language basics.

GCC

It's almost the same situation in GCC 4.4.1, with few exceptions.

```
                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
```

Here we see another *name mangling* style, specific to GNU[100] standards. It's also can be noted that pointer to object is passed as first function argument — hiddenly from programmer, of course.

First constructor:

```
                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
```

What it does is just writes two numbers using pointer passed in first (and single) argument.

Second constructor:

```
                public _ZN1cC1Eii
_ZN1cC1Eii      proc near
```

[100] One more document about different compilers name mangling types: http://www.agner.org/optimize/calling_conventions.pdf

```
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
```

This is a function, analog of which could be looks like:

```
void ZN1cC1Eii (int *obj, int a, int b)
{
  *obj=a;
  *(obj+1)=b;
};
```

...and that's completely predictable.

Now `dump()` function:

```
                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
```

This function in its *internal representation* has sole argument, used as pointer to the object (*this*).

Thus, if to base our judgment on these simple examples, the difference between MSVC and GCC is style of function names encoding (*name mangling*) and passing pointer to object (via `ECX` register or via first argument).

1.17.2 Class inheritance in C++

It can be said about inherited classes that it's simple structure we already considered, but extending in inherited classes.

Let's take simple example:

```
#include <stdio.h>

class object
{
    public:
        int color;
        object() { };
        object (int color) { this->color=color; };
        void print_color() { printf ("color=%d\n", color); };
};

class box : public object
{
    private:
        int width, height, depth;
    public:
        box(int color, int width, int height, int depth)
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is box. color=%d, width=%d, height=%d, depth=%d\n", color, width, height, depth);
        };
};

class sphere : public object
{
private:
    int radius;
public:
    sphere(int color, int radius)
    {
        this->color=color;
        this->radius=radius;
    };
    void dump()
    {
        printf ("this is sphere. color=%d, radius=%d\n", color, radius);
    };
};

int main()
{
    box b(1, 10, 20, 30);
    sphere s(2, 40);

    b.print_color();
    s.print_color();

    b.dump();
    s.dump();

    return 0;
};
```

Let's investigate generated code of `dump()` functions/methods and also `object::print_color()`, let's see

memory layout for structures-objects (as of 32-bit code).

So, `dump()` methods for several classes, generated by MSVC 2008 with `/Ox` and `/Ob0` options [101]

Listing 1.98: Optimizing MSVC 2008 /Ob0

```
??_C@_09GCEDOLPA@color?$DN?$CFd?6?$AA@ DB 'color=%d', 0aH, 00H ; 'string'
?print_color@object@@QAEXXZ PROC                        ; object::print_color, COMDAT
; _this$ = ecx
        mov     eax, DWORD PTR [ecx]
        push    eax

; 'color=%d', 0aH, 00H
        push    OFFSET ??_C@_09GCEDOLPA@color?$DN?$CFd?6?$AA@
        call    _printf
        add     esp, 8
        ret     0
?print_color@object@@QAEXXZ ENDP                        ; object::print_color
```

Listing 1.99: Optimizing MSVC 2008 /Ob0

```
?dump@box@@QAEXXZ PROC                                  ; box::dump, COMDAT
; _this$ = ecx
        mov     eax, DWORD PTR [ecx+12]
        mov     edx, DWORD PTR [ecx+8]
        push    eax
        mov     eax, DWORD PTR [ecx+4]
        mov     ecx, DWORD PTR [ecx]
        push    edx
        push    eax
        push    ecx

; 'this is box. color=%d, width=%d, height=%d, depth=%d', 0aH, 00H ; 'string'
        push    OFFSET ??_C@_0DG@NCNGAADL@this?5is?5box?4?5color?$DN?$CFd?0?5width?$DN?$CFd?0@
        call    _printf
        add     esp, 20                                 ; 00000014H
        ret     0
?dump@box@@QAEXXZ ENDP                                  ; box::dump
```

Listing 1.100: Optimizing MSVC 2008 /Ob0

```
?dump@sphere@@QAEXXZ PROC                               ; sphere::dump, COMDAT
; _this$ = ecx
        mov     eax, DWORD PTR [ecx+4]
        mov     ecx, DWORD PTR [ecx]
        push    eax
        push    ecx

; 'this is sphere. color=%d, radius=%d', 0aH, 00H
        push    OFFSET ??_C@_0CF@EFEDJLDC@this?5is?5sphere?4?5color?$DN?$CFd?0?5radius@
        call    _printf
        add     esp, 12                                 ; 0000000cH
        ret     0
?dump@sphere@@QAEXXZ ENDP                               ; sphere::dump
```

So, here is memory layout:
(base class *object*)

offset	description
+0x0	int color

(inherited classes)
box:

[101] `/Ob0` options mean inline expansion disabling, because, function inlining right into the code where the function is called will make our experiment harder

offset	description
+0x0	int color
+0x4	int width
+0x8	int height
+0xC	int depth

sphere:

offset	description
+0x0	int color
+0x4	int radius

Let's see `main()` function body:

Listing 1.101: Optimizing MSVC 2008 /Ob0

```
PUBLIC  _main
_TEXT   SEGMENT
_s$ = -24                                               ; size = 8
_b$ = -16                                               ; size = 16
_main   PROC
        sub     esp, 24                                 ; 00000018H
        push    30                                      ; 0000001eH
        push    20                                      ; 00000014H
        push    10                                      ; 0000000aH
        push    1
        lea     ecx, DWORD PTR _b$[esp+40]
        call    ??0box@@QAE@HHHH@Z                      ; box::box
        push    40                                      ; 00000028H
        push    2
        lea     ecx, DWORD PTR _s$[esp+32]
        call    ??0sphere@@QAE@HH@Z                     ; sphere::sphere
        lea     ecx, DWORD PTR _b$[esp+24]
        call    ?print_color@object@@QAEXXZ             ; object::print_color
        lea     ecx, DWORD PTR _s$[esp+24]
        call    ?print_color@object@@QAEXXZ             ; object::print_color
        lea     ecx, DWORD PTR _b$[esp+24]
        call    ?dump@box@@QAEXXZ                       ; box::dump
        lea     ecx, DWORD PTR _s$[esp+24]
        call    ?dump@sphere@@QAEXXZ                    ; sphere::dump
        xor     eax, eax
        add     esp, 24                                 ; 00000018H
        ret     0
_main   ENDP
```

Inherited classes should always add their fields after base classes' fields, so to make possible for base class methods to work with their fields.

When `object::print_color()` method is called, a pointers to both *box* object and *sphere* object are passed as `this`, it can work with these objects easily because *color* field in these objects is always at the pinned address (at *+0x0* offset).

It can be said, `object::print_color()` method is agnostic in relation to input object type as long as fields will be *pinned* at the same addresses, and this condition is always true.

And if you create inherited class of *box* class, for example, compiler will add new fields after *depth* field, leaving *box* class fields at the pinned addresses.

Thus, `box::dump()` method will work fine accessing *color*/*width*/*height*/*depths* fields always pinned on known addresses.

GCC-generated code is almost the same, with the sole exception of `this` pointer passing (as it was described above, it passing as first argument instead of `ECX` registers.

1.17.3 Encapsulation in C++

Encapsulation is data hiding in *private* sections of class, for example, to allow access to them only from this class methods, but no more than.

However, are there any marks in code about the fact that some field is private and some other — not?

No, there are no such marks.

Let's try simple example:

```
#include <stdio.h>

class box
{
    private:
        int color, width, height, depth;
    public:
        box(int color, int width, int height, int depth)
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is box. color=%d, width=%d, height=%d, depth=%d\n", color, width, height, depth);
        };
};
```

Let's compile it again in MSVC 2008 with `/Ox` and `/Ob0` options and let's see `box::dump()` method code:

```
?dump@box@@QAEXXZ PROC                                  ; box::dump, COMDAT
; _this$ = ecx
        mov     eax, DWORD PTR [ecx+12]
        mov     edx, DWORD PTR [ecx+8]
        push    eax
        mov     eax, DWORD PTR [ecx+4]
        mov     ecx, DWORD PTR [ecx]
        push    edx
        push    eax
        push    ecx
; 'this is box. color=%d, width=%d, height=%d, depth=%d', 0aH, 00H
        push    OFFSET ??_C@_0DG@NCNGAADL@this?5is?5box?4?5color?$DN?$CFd?0?5width?$DN?$CFd?0@
        call    _printf
        add     esp, 20                                 ; 00000014H
        ret     0
?dump@box@@QAEXXZ ENDP                                  ; box::dump
```

Here is a memory layout of the class:

offset	description
+0x0	int color
+0x4	int width
+0x8	int height
+0xC	int depth

All fields are private and not allowed to access from any other functions, but, knowing this layout, can we create a code modifying these fields?

So I added `hack_oop_encapsulation()` function, which, if has the body as follows, won't compile:

```
void hack_oop_encapsulation(class box * o)
{
    o->width=1; // that code can't be compiled: "error C2248: 'box::width' : cannot access private member
    declared in class 'box'"
};
```

Nevertheless, if to cast *box* type to *pointer to int array*, and if to modify array of *int*-s we got, then we have success.

```
void hack_oop_encapsulation(class box * o)
{
    unsigned int *ptr_to_object=reinterpret_cast<unsigned int*>(o);
    ptr_to_object[1]=123;
};
```

This functions' code is very simple — it can be said, the function taking pointer to array of *int*-s on input and writing *123* to the second *int*:

```
?hack_oop_encapsulation@@YAXPAVbox@@@Z PROC              ; hack_oop_encapsulation
        mov     eax, DWORD PTR _o$[esp-4]
        mov     DWORD PTR [eax+4], 123                  ; 0000007bH
        ret     0
?hack_oop_encapsulation@@YAXPAVbox@@@Z ENDP              ; hack_oop_encapsulation
```

Let's check, how it works:

```
int main()
{
    box b(1, 10, 20, 30);

    b.dump();

    hack_oop_encapsulation(&b);

    b.dump();

    return 0;
};
```

Let's run:

```
this is box. color=1, width=10, height=20, depth=30
this is box. color=1, width=123, height=20, depth=30
```

We see, encapsulation is just class fields protection only on compiling stage. C++ compiler won't allow to generate a code modifying protected fields straightforwardly, nevertheless, it's possible using *dirty hacks*.

1.17.4 Multiple inheritance in C++

Multiple inheritance is a class creation which inherits fields and methods from two or more classes.

Let's write simple example again:

```
#include <stdio.h>

class box
{
    public:
        int width, height, depth;
        box() { };
        box(int width, int height, int depth)
        {
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is box. width=%d, height=%d, depth=%d\n", width, height, depth);
        };
        int get_volume()
        {
            return width * height * depth;
        };
};

class solid_object
{
    public:
        int density;
        solid_object() { };
        solid_object(int density)
        {
            this->density=density;
        };
        int get_density()
        {
            return density;
        };
        void dump()
        {
            printf ("this is solid_object. density=%d\n", density);
        };
};

class solid_box: box, solid_object
{
    public:
        solid_box (int width, int height, int depth, int density)
        {
            this->width=width;
            this->height=height;
            this->depth=depth;
            this->density=density;
        };
        void dump()
        {
            printf ("this is solid_box. width=%d, height=%d, depth=%d, density=%d\n", width, height, depth,
    density);
        };
        int get_weight() { return get_volume() * get_density(); };
};

int main()
```

```
{
    box b(10, 20, 30);
    solid_object so(100);
    solid_box sb(10, 20, 30, 3);

    b.dump();
    so.dump();
    sb.dump();
    printf ("%d\n", sb.get_weight());

    return 0;
};
```

Let's compile it in MSVC 2008 with `/Ox` and `/Ob0` options and let's see `box::dump()`, `solid_object::dump()` and `solid_box::dump()` methods code:

Listing 1.102: Optimizing MSVC 2008 /Ob0

```
?dump@box@@QAEXXZ PROC                                  ; box::dump, COMDAT
; _this$ = ecx
        mov     eax, DWORD PTR [ecx+8]
        mov     edx, DWORD PTR [ecx+4]
        push    eax
        mov     eax, DWORD PTR [ecx]
        push    edx
        push    eax
; 'this is box. width=%d, height=%d, depth=%d', 0aH, 00H
        push    OFFSET ??_C@_0CM@DIKPHDFI@this?5is?5box?4?5width?$DN?$CFd?0?5height?$DN?$CFd@
        call    _printf
        add     esp, 16                                 ; 00000010H
        ret     0
?dump@box@@QAEXXZ ENDP                                  ; box::dump
```

Listing 1.103: Optimizing MSVC 2008 /Ob0

```
?dump@solid_object@@QAEXXZ PROC                         ; solid_object::dump, COMDAT
; _this$ = ecx
        mov     eax, DWORD PTR [ecx]
        push    eax
; 'this is solid_object. density=%d', 0aH
        push    OFFSET ??_C@_0CC@KICFJINL@this?5is?5solid_object?4?5density?$DN?$CFd@
        call    _printf
        add     esp, 8
        ret     0
?dump@solid_object@@QAEXXZ ENDP                         ; solid_object::dump
```

Listing 1.104: Optimizing MSVC 2008 /Ob0

```
?dump@solid_box@@QAEXXZ PROC                            ; solid_box::dump, COMDAT
; _this$ = ecx
        mov     eax, DWORD PTR [ecx+12]
        mov     edx, DWORD PTR [ecx+8]
        push    eax
        mov     eax, DWORD PTR [ecx+4]
        mov     ecx, DWORD PTR [ecx]
        push    edx
        push    eax
        push    ecx
; 'this is solid_box. width=%d, height=%d, depth=%d, density=%d', 0aH
        push    OFFSET ??_C@_0DO@HNCNIHNN@this?5is?5solid_box?4?5width?$DN?$CFd?0?5hei@
        call    _printf
        add     esp, 20                                 ; 00000014H
        ret     0
?dump@solid_box@@QAEXXZ ENDP                            ; solid_box::dump
```

So, the memory layout for all three classes is:

box class:

offset	description
+0x0	width
+0x4	height
+0x8	depth

solid_object class:

offset	description
+0x0	density

It can be said, *solid_box* class memory layout will be *united*:
solid_box class:

offset	description
+0x0	width
+0x4	height
+0x8	depth
+0xC	density

The code of `box::get_volume()` and `solid_object::get_density()` methods is trivial:

Listing 1.105: Optimizing MSVC 2008 /Ob0

```
?get_volume@box@@QAEHXZ PROC                            ; box::get_volume, COMDAT
; _this$ = ecx
        mov     eax, DWORD PTR [ecx+8]
        imul    eax, DWORD PTR [ecx+4]
        imul    eax, DWORD PTR [ecx]
        ret     0
?get_volume@box@@QAEHXZ ENDP                            ; box::get_volume
```

Listing 1.106: Optimizing MSVC 2008 /Ob0

```
?get_density@solid_object@@QAEHXZ PROC                  ; solid_object::get_density, COMDAT
; _this$ = ecx
        mov     eax, DWORD PTR [ecx]
        ret     0
?get_density@solid_object@@QAEHXZ ENDP                  ; solid_object::get_density
```

But the code of `solid_box::get_weight()` method is much more interesting:

Listing 1.107: Optimizing MSVC 2008 /Ob0

```
?get_weight@solid_box@@QAEHXZ PROC                      ; solid_box::get_weight, COMDAT
; _this$ = ecx
        push    esi
        mov     esi, ecx
        push    edi
        lea     ecx, DWORD PTR [esi+12]
        call    ?get_density@solid_object@@QAEHXZ       ; solid_object::get_density
        mov     ecx, esi
        mov     edi, eax
        call    ?get_volume@box@@QAEHXZ                 ; box::get_volume
        imul    eax, edi
        pop     edi
        pop     esi
        ret     0
?get_weight@solid_box@@QAEHXZ ENDP                      ; solid_box::get_weight
```

`get_weight()` just calling two methods, but for `get_volume()` it just passing pointer to `this`, and for `get_density()` it passing pointer to `this` shifted by *12* (or *0xC*) bytes, and there, in `solid_box` class memory layout, fields of `solid_object` class are beginning.

Thus, `solid_object::get_density()` method will believe it is working with usual `solid_object` class, and `box::get_volume()` method will work with its three fields, believing this is usual object of `box` class.

Thus, we can say, an object of some class, inheriting from several classes, representing in memory *united* class, containing all inherited fields. And each inherited method called with a pointer to corresponding structure's part passed.

1.17.5 C++ virtual methods

Yet another simple example:

```
#include <stdio.h>

class object
{
    public:
        int color;
        object() { };
        object (int color) { this->color=color; };
        virtual void dump()
        {
            printf ("color=%d\n", color);
        };
};

class box : public object
{
    private:
        int width, height, depth;
    public:
        box(int color, int width, int height, int depth)
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is box. color=%d, width=%d, height=%d, depth=%d\n", color, width, height, depth);
        };
};

class sphere : public object
{
    private:
        int radius;
    public:
        sphere(int color, int radius)
        {
            this->color=color;
            this->radius=radius;
        };
        void dump()
        {
            printf ("this is sphere. color=%d, radius=%d\n", color, radius);
        };
};

int main()
{
    box b(1, 10, 20, 30);
    sphere s(2, 40);

    object *o1=&b;
    object *o2=&s;

    o1->dump();
    o2->dump();
    return 0;
};
```

Class *object* has virtual method `dump()`, being replaced in *box* and *sphere* class-inheritors.

If in some environment, where it's not known what type has object, as in `main()` function in example, a virtual method `dump()` is called, somewhere, the information about its type should be stored, so to call relevant virtual method.

Let's compile it in MSVC 2008 with `/Ox` and `/Ob0` options and let's see `main()` function code:

```
_s$ = -32                                               ; size = 12
_b$ = -20                                               ; size = 20
_main   PROC
        sub     esp, 32                                 ; 00000020H
        push    30                                      ; 0000001eH
        push    20                                      ; 00000014H
        push    10                                      ; 0000000aH
        push    1
        lea     ecx, DWORD PTR _b$[esp+48]
        call    ??0box@@QAE@HHHH@Z                      ; box::box
        push    40                                      ; 00000028H
        push    2
        lea     ecx, DWORD PTR _s$[esp+40]
        call    ??0sphere@@QAE@HH@Z                     ; sphere::sphere
        mov     eax, DWORD PTR _b$[esp+32]
        mov     edx, DWORD PTR [eax]
        lea     ecx, DWORD PTR _b$[esp+32]
        call    edx
        mov     eax, DWORD PTR _s$[esp+32]
        mov     edx, DWORD PTR [eax]
        lea     ecx, DWORD PTR _s$[esp+32]
        call    edx
        xor     eax, eax
        add     esp, 32                                 ; 00000020H
        ret     0
_main   ENDP
```

Pointer to `dump()` function is taken somewhere from object. Where the address of new method would be written there? Only somewhere in constructors: there are no other place, because, nothing more is called in `main()` function. [102]

Let's see *box* class constructor's code:

```
??_R0?AVbox@@@8 DD FLAT:??_7type_info@@6B@              ; box 'RTTI Type Descriptor'
        DD      00H
        DB      '.?AVbox@@', 00H

??_R1A@?0A@EA@box@@8 DD FLAT:??_R0?AVbox@@@8            ; box::'RTTI Base Class Descriptor at (0,-1,0,64)'
        DD      01H
        DD      00H
        DD      0ffffffffH
        DD      00H
        DD      040H
        DD      FLAT:??_R3box@@8

??_R2box@@8 DD  FLAT:??_R1A@?0A@EA@box@@8               ; box::'RTTI Base Class Array'
        DD      FLAT:??_R1A@?0A@EA@object@@8

??_R3box@@8 DD  00H                                     ; box::'RTTI Class Hierarchy Descriptor'
        DD      00H
        DD      02H
        DD      FLAT:??_R2box@@8

??_R4box@@6B@ DD 00H                                    ; box::'RTTI Complete Object Locator'
        DD      00H
        DD      00H
        DD      FLAT:??_R0?AVbox@@@8
        DD      FLAT:??_R3box@@8

??_7box@@6B@ DD FLAT:??_R4box@@6B@                      ; box::'vftable'
```

[102] About pointers to functions, read more in relevant section:1.19

```
        DD      FLAT:?dump@box@@UAEXXZ

_color$ = 8                                             ; size = 4
_width$ = 12                                            ; size = 4
_height$ = 16                                           ; size = 4
_depth$ = 20                                            ; size = 4
??0box@@QAE@HHHH@Z PROC                                 ; box::box, COMDAT
; _this$ = ecx
        push    esi
        mov     esi, ecx
        call    ??0object@@QAE@XZ                       ; object::object
        mov     eax, DWORD PTR _color$[esp]
        mov     ecx, DWORD PTR _width$[esp]
        mov     edx, DWORD PTR _height$[esp]
        mov     DWORD PTR [esi+4], eax
        mov     eax, DWORD PTR _depth$[esp]
        mov     DWORD PTR [esi+16], eax
        mov     DWORD PTR [esi], OFFSET ??_7box@@6B@
        mov     DWORD PTR [esi+8], ecx
        mov     DWORD PTR [esi+12], edx
        mov     eax, esi
        pop     esi
        ret     16                                      ; 00000010H
??0box@@QAE@HHHH@Z ENDP                                 ; box::box
```

Here we see slightly different memory layout: the first field is a pointer to some table `box::'vftable'` (name was set by MSVC compiler).

In this table we see a link to the table named `box::'RTTI Complete Object Locator'` and also a link to `box::dump()` method. So this is named virtual methods table and RTTI[103]. Table of virtual methods contain addresses of methods and RTTI table contain information about types. By the way, RTTI-tables are the tables enumerated while calling to *dynamic_cast* and *typeid* in C++. You can also see here class name as plain text string. Thus, some method of base *object* class may call virtual method *object::dump()*, which in turn, will call a method of inherited class, because that information is present right in the object's structure.

Some CPU time needed for enumerating these tables and finding right virtual method address, thus virtual methods are widely considered as slightly slower than usual methods.

In GCC-generated code RTTI-tables constructed slightly differently.

[103] Run-time type information

1.18 Unions

1.18.1 Pseudo-random number generator example

If we need float random numbers from 0 to 1, the most simplest thing is to use random numbers generator like Mersenne twister producing random 32-bit values in DWORD form, transform this value to *float* and then divide it by `RAND_MAX` (*0xffffffff* in our case) — value we got will be in 0..1 interval.

But as we know, division operation is almost always very slow. Will it be possible to get rid of it, as in case of division by multiplication? 1.12

Let's recall what float number consisted of: sign bit, significand bits and exponent bits. We need just to store random bits to all significand bits for getting random float number!

Exponent cannot be zero (number will be denormalized in this case), so we will store *01111111* to exponent — this mean exponent will be 1. Then fill significand with random bits, set sign bit to 0 (which mean positive number) and voilà. Generated numbers will be in 1 to 2 interval, so we also should subtract 1 from it.

Very simple linear congruential random numbers generator is used in my example[104], producing 32-bit numbers. The PRNG initializing by current time in UNIX-style.

Then, *float* type represented as *union* — that is the C/C++ construction enabling us to interpret piece of memory differently typed. In our case, we are able to create a variable of `union` type and then access to it as it's *float* or as it's *uint32_t*. It can be said, it's just a hack. A dirty one.

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

union uint32_t_float
{
    uint32_t i;
    float f;
};

// from the Numerical Recipes book
const uint32_t RNG_a=1664525;
const uint32_t RNG_c=1013904223;

int main()
{
    uint32_t_float tmp;

    uint32_t RNG_state=time(NULL); // initial seed
    for (int i=0; i<100; i++)
    {
        RNG_state=RNG_state*RNG_a+RNG_c;
        tmp.i=RNG_state & 0x007fffff | 0x3F800000;
        float x=tmp.f-1;
        printf ("%f\n", x);
    };
    return 0;
};
```

MSVC 2010 (`/Ox`):

```
$SG4232    DB    '%f', 0aH, 00H

__real@3ff0000000000000 DQ 03ff0000000000000r    ; 1

tv140 = -4                          ; size = 4
_tmp$ = -4                          ; size = 4
_main    PROC
    push   ebp
    mov    ebp, esp
```

[104] idea was taken from: http://xor0110.wordpress.com/2010/09/24/how-to-generate-floating-point-random-numbers-efficiently

```
    and    esp, -64               ; ffffffc0H
    sub    esp, 56                ; 00000038H
    push   esi
    push   edi
    push   0
    call   __time64
    add    esp, 4
    mov    esi, eax
    mov    edi, 100               ; 00000064H
$LN3@main:

; let's generate random 32-bit number

    imul   esi, 1664525           ; 0019660dH
    add    esi, 1013904223        ; 3c6ef35fH
    mov    eax, esi

; leave bits for significand only

    and    eax, 8388607           ; 007fffffH

; set exponent to 1

    or     eax, 1065353216        ; 3f800000H

; store this value as int

    mov    DWORD PTR _tmp$[esp+64], eax
    sub    esp, 8

; load this value as float

    fld    DWORD PTR _tmp$[esp+72]

; subtract one from it

    fsub   QWORD PTR __real@3ff0000000000000
    fstp   DWORD PTR tv140[esp+72]
    fld    DWORD PTR tv140[esp+72]
    fstp   QWORD PTR [esp]
    push   OFFSET $SG4232
    call   _printf
    add    esp, 12                ; 0000000cH
    dec    edi
    jne    SHORT $LN3@main
    pop    edi
    xor    eax, eax
    pop    esi
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
END
```

GCC producing very same code.

1.19 Pointers to functions

Pointer to function, as any other pointer, is just an address of function beginning in its code segment.

It is often used in callbacks [105].

Well-known examples are:

- `qsort()`[106], `atexit()`[107] from the standard C library;
- signals in *NIX OS[108];
- thread starting: `CreateThread()` (win32), `pthread_create()` (POSIX);
- a lot of win32 functions, e.g. `EnumChildWindows()`[109].

So, `qsort()` function is a C/C++ standard library quicksort implemenation. The functions is able to sort anything, any types of data, if you have a function for two elements comparison and `qsort()` is able to call it.

The comparison function can be defined as:

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

Let's use slightly modified example I found here:

```
/* 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;
}
```

Let's compile it in MSVC 2010 (I omitted some parts for the sake of brefity) with `/Ox` option:

Listing 1.108: Optimizing MSVC 2010

```
__a$ = 8            ; size = 4
__b$ = 12           ; size = 4
_comp      PROC
```

[105] http://en.wikipedia.org/wiki/Callback_(computer_science)
[106] http://en.wikipedia.org/wiki/Qsort_(C_standard_library)
[107] http://www.opengroup.org/onlinepubs/009695399/functions/atexit.html
[108] http://en.wikipedia.org/wiki/Signal.h
[109] http://msdn.microsoft.com/en-us/library/ms633494(VS.85).aspx

```
    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

...
```

Nothing surprising so far. As a fourth argument, an address of label `_comp` is passed, that's just a place where function `comp()` located.

How `qsort()` calling it?

Let's take a look into this function located in MSVCR80.DLL (a MSVC DLL module with C standard library functions):

Listing 1.109: MSVCR80.DLL

```
.text:7816CBF0 ; void __cdecl qsort(void *, unsigned int, unsigned int, int (__cdecl *)(const void *, const
    void *))
.text:7816CBF0                 public _qsort
.text:7816CBF0 _qsort          proc near
.text:7816CBF0
.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` — is fourth function argument. Here the control is just passed to the address in `comp`. Before it, two arguments prepared for `comp()`. Its result is checked after its execution.

That's why it's dangerous to use pointers to functions. First of all, if you call `qsort()` with incorrect pointer to function, `qsort()` may pass control to incorrect place, process may crash and this bug will be hard to find.

Second reason is that callback function types should comply strictly, calling wrong function with wrong arguments of wrong types may lead to serious problems, however, process crashing is not a big problem — big problem is to determine a reason of crashing — because compiler may be silent about potential trouble while compiling.

1.19.1 GCC

Not a big difference:

Listing 1.110: GCC

```
                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
```

`comp()` function:

```
                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
```

`qsort()` implementation is located in `libc.so.6` and it is in fact just a wrapper for `qsort_r()`. It will call then `quicksort()`, where our defined function will be called via passed pointer:

Listing 1.111: (File libc.so.6, glibc version — 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]
...
```

1.20 SIMD

SIMD is just acronym: *Single Instruction, Multiple Data*.

As it's said, it's multiple data processing using only one instruction.

Just as FPU, that CPU subsystem looks like separate processor inside x86.

SIMD began as MMX in x86. 8 new 64-bit registers appeared: MM0-MM7.

Each MMX register may hold 2 32-bit values, 4 16-bit values or 8 bytes. For example, it is possible to add 8 8-bit values (bytes) simultaneously by adding two values in MMX-registers.

One simple example is graphics editor, representing image as a two dimensional array. When user change image brightness, the editor should add some coefficient to each pixel value, or to subtract. For the sake of brevity, our image may be grayscale and each pixel defined by one 8-bit byte, then it's possible to change brightness of 8 pixels simultaneously.

When MMX appeared, these registers was actually located in FPU registers. It was possible to use either FPU or MMX at the same time. One might think, Intel saved on transistors, but in fact, the reason of such symbiosis is simpler — older operation system may not aware of additional CPU registers wouldn't save them at the context switching, but will save FPU registers. Thus, MMX-enabled CPU + old operation system + process using MMX = that all will work together.

SSE — is extension of SIMD registers up to 128 bits, now separately from FPU.

AVX — another extension to 256 bits.

Now about practical usage.

Of course, memory copying (`memcpy`), memory comparing (`memcmp`) and so on.

One more example: we got DES encryption algorithm, it takes 64-bit block, 56-bit key, encrypt block and produce 64-bit result. DES algorithm may be considered as a very large electronic circuit, with wires and AND/OR/NOT gates.

Bitslice DES[110] — is an idea of processing group of blocks and keys simultaneously. Let's say, variable of type *unsigned int* on x86 may hold up to 32 bits, so, it's possible to store there intermediate results for 32 blocks-keys pairs simultaneously, using 64+56 variables of *unsigned int* type.

I wrote an utility to brute-force Oracle RDBMS passwords/hashes (ones based on DES), slightly modified bitslice DES algorithm for SSE2 and AVX — now it's possible to encrypt 128 or 256 block-keys pairs simultaneously.

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

1.20.1 Vectorization

Vectorization[111], for example, is when you have a loop taking couple of arrays at input and producing one array. Loop body takes values from input arrays, do something and put result into output array. It's important that there is only one single operation applied to each element. Vectorization — is to process several elements simultaneously.

For example:

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

This fragment of code takes elements from A and B, multiplies them and save result into C.

If each array element we have is 32-bit *int*, then it's possible to load 4 elements from A into 128-bit XMM-register, from B to another XMM-registers, and by executing *PMULLD* (*Multiply Packed Signed Dword Integers and Store Low Result*)and `PMULHW` (*Multiply Packed Signed Integers and Store High Result*), it's possible to get 4 64-bit products[112] at once.

Thus, loop body count is $1024/4$ instead of 1024, that's 4 times less and, of course, faster.

Some compilers can do vectorization automatically in some simple cases, e.g., Intel C++[113].

I wrote tiny function:

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

[111] Wikipedia: vectorization

[112] multiplication result

[113] More about Intel C++ automatic vectorization: Excerpt: Effective Automatic Vectorization

```
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++

Let's compile it with Intel C++ 11.1.051 win32:

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

We got (in IDA 5):

```
; 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]
                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
```

SSE2-related instructions are:

- MOVDQU (*Move Unaligned Double Quadword*) — it just load 16 bytes from memory into XMM-register.
- PADDD (*Add Packed Integers*) — adding 4 pairs of 32-bit numbers and leaving result in first operand. By the way, no exception raised in case of overflow and no flags will be set, just low 32-bit of result will be stored. If one of PADDD operands — address of value in memory, address should be aligned by 16-byte border. If it's not aligned, exception will be raised [114].

[114] More about data aligning: Wikipedia: Data structure alignment

- MOVDQA (*Move Aligned Double Quadword*) — the same as MOVDQU, but requires address of value in memory to be aligned by 16-bit border. If it's not aligned, exception will be raised. MOVDQA works faster than MOVDQU, but requires aforesaid.

So, these SSE2-instructions will be executed only in case if there are more 4 pairs to work on plus pointer ar3 is aligned on 16-byte border.

More than that, if ar2 is aligned on 16-byte border too, this fragment of code will be executed:

```
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
```

Otherwise, value from ar2 will be loaded to XMM0 using MOVDQU, it doesn't require aligned pointer, but may work slower:

```
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
```

In all other cases, non-SSE2 code will be executed.

GCC

GCC may also vectorize in some simple cases[115], if to use -O3 option and to turn on SSE2 support: -msse2.

What we got (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
                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]
```

[115] More about GCC vectorization support: http://gcc.gnu.org/projects/tree-ssa/vectorization.html

```
                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
```

Almost the same, however, not as meticulously as Intel C++ doing it.

1.20.2 SIMD `strlen()` implementation

It should be noted that SIMD-instructions may be inserted into C/C++ code via special macros[116]. As of MSVC, some of them are located in `intrin.h` file.

It is possible to implement `strlen()` function[117] using SIMD-instructions, working 2-2.5 times faster than usual implementation. This function will load 16 characters into XMM-register and check each against zero.

```
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;

    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);
```

[116] MSDN: MMX, SSE, and SSE2 Intrinsics

[117] strlen() — standard C library function for calculating string length

```
    };

    return len;
}
```

(the example is based on source code from there).

Let's compile in MSVC 2010 with `/Ox` option:

```
_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
```

First of all, we check `str` pointer, if it's aligned by 16-byte border. If not, let's call usual `strlen()` implementation.

Then, load next 16 bytes into `XMM1` register using `MOVDQA` instruction.

Observant reader might ask, why MOVDQU cannot be used here, because it can load data from the memory regardless the fact if the pointer aligned or not.

Yes, it might be done in this way: if pointer is aligned, load data using MOVDQA, if not — use slower MOVDQU.

But here we are may stick into hard to notice caveat:

In Windows NT line of operation systems[118], but not limited to it, memory allocated by pages of 4 KiB (4096 bytes). Each win32-process has ostensibly 4 GiB, but in fact, only some parts of address space are connected to real physical memory. If the process accessing to the absent memory block, exception will be raised. That's how virtual memory works[119].

So, some function loading 16 bytes at once, may step over a border of allocated memory block. Let's consider, OS allocated 8192 (0x2000) bytes at the address 0x008c0000. Thus, the block is the bytes starting from address 0x008c0000 to 0x008c1fff inclusive.

After that block, that is, starting from address 0x008c2008 there are nothing at all, e.g., OS not allocated any memory there. Attempt to access a memory starting from that address will raise exception.

And let's consider, the program holding some string containing 5 characters almost at the end of block, and that's not a crime.

0x008c1ff8	'h'
0x008c1ff9	'e'
0x008c1ffa	'l'
0x008c1ffb	'l'
0x008c1ffc	'o'
0x008c1ffd	'\x00'
0x008c1ffe	random noise
0x008c1fff	random noise

So, in usual conditions the program calling strlen() passing it a pointer to string 'hello' lying in memory at address 0x008c1ff8. strlen() will read one byte at a time until 0x008c1ffd, where zero-byte, and so here it will stop working.

Now if we implement own strlen() reading 16 byte at once, starting at any address, will it be alligned or not, MOVDQU may attempt to load 16 bytes at once at address 0x008c1ff8 up to 0x008c2008, and then exception will be raised. That's the situation to be avoided, of course.

So then we'll work only with the addresses aligned by 16 byte border, what in combination with a knowledge of operation system page size is usually aligned by 16 byte too, give us some warranty our function will not read from unallocated memory.

Let's back to our function.

_mm_setzero_si128() — is a macro generating pxor xmm0, xmm0 — instruction just clear /XMMZERO register

_mm_load_si128() — is a macro for MOVDQA, it just loading 16 bytes from the address in XMM1.

_mm_cmpeq_epi8() — is a macro for PCMPEQB, is an instruction comparing two XMM-registers bytewise.

And if some byte was equals to other, there will be 0xff at this place in result or 0 if otherwise.

For example.

```
XMM1: 11223344556677880000000000000000
XMM0: 11ab3444007877881111111111111111
```

After pcmpeqb xmm1, xmm0 execution, XMM1 register will contain:

```
XMM1: ff0000ff0000ffff0000000000000000
```

In our case, this instruction comparing each 16-byte block with the block of 16 zero-bytes, was set in XMM0 by pxor xmm0, xmm0.

The next macro is _mm_movemask_epi8() — that is PMOVMSKB instruction.

It is very useful if to use it with PCMPEQB.

[118] Windows NT, 2000, XP, Vista, 7, 8

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

`pmovmskb eax, xmm1`

This instruction will set first `EAX` bit into 1 if most significant bit of the first byte in `XMM1` is 1. In other words, if first byte of `XMM1` register is 0xff, first `EAX` bit will be set to 1 too.

If second byte in `XMM1` register is 0xff, then second `EAX` bit will be set to 1 too. In other words, the instruction is answer to the question *which bytes in `XMM1` are 0xff?* And will prepare 16 bits in `EAX`. Other `EAX` bits will be cleared.

By the way, do not forget about this feature of our algorithm:

There might be 16 bytes on input like `hello\x00garbage\x00ab`

It's a `'hello'` string, terminating zero after and some random noise in memory.

If we load these 16 bytes into `XMM1` and compare them with zeroed `XMM0`, we will get something like (I use here order from MSB[120] to LSB[121]):

```
XMM1: 0000ff00000000000000ff0000000000
```

This mean, the instruction found two zero bytes, and that's not surprising.

`PMOVMSKB` in our case will prepare `EAX` like (in binary representation): *0010000000100000b*.

Obviously, our function should consider only first zero and ignore others.

The next instruction — `BSF` (*Bit Scan Forward*). This instruction find first bit set to 1 and stores its position into first operand.

```
EAX=00100000001000000b
```

After `bsf eax, eax` instruction execution, `EAX` will contain 5, this mean, 1 found at 5th bit position (starting from zero).

MSVC has a macro for this instruction: `_BitScanForward`.

Now it's simple. If zero byte found, its position added to what we already counted and now we have ready to return result.

Almost all.

By the way, it's also should be noted, MSVC compiler emitted two loop bodies side by side, for optimization.

By the way, SSE 4.2 (appeared in Intel Core i7) offers more instructions where these string manipulations might be even easier: http://www.strchr.com/strcmp_and_strlen_using_sse_4.2

[120] most significant bit

[121] least significant bit

1.21 64 bits

1.21.1 x86-64

It's a 64-bit extension to x86-architecture.

From the reverse engineer's perspective, most important differences are:

- Almost all registers (except FPU and SIMD) are extended to 64 bits and got r- prefix. 8 additional registers added. Now general purpose registers are: `rax`, `rbx`, `rcx`, `rdx`, `rbp`, `rsp`, `rsi`, `rdi`, `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`, `r15`.

  It's still possible to access to *older* register parts as usual. For example, it's possible to access lower 32-bit part of `RAX` using `EAX`.

  New `r8-r15` registers also has its *lower parts*: `r8d-r15d` (lower 32-bit parts), `r8w-r15w` (lower 16-bit parts), `r8b-r15b` (lower 8-bit parts).

  SIMD-registers number are doubled: from 8 to 16: `XMM0-XMM15`.

- In Win64, function calling convention is slightly different, somewhat resembling fastcall 2.5.3. First 4 arguments stored in `RCX`, `RDX`, `R8`, `R9` registers, others — in stack. Caller function should also allocate 32 bytes so the callee may save there 4 first arguments and use these registers for own needs. Short functions may use arguments just from registers, but larger may save their values into stack.

  See also section about calling conventions 2.5.

- C *int* type is still 32-bit for compatibility.

- All pointers are 64-bit now.

Since now registers number are doubled, compilers has more space now for maneuvering calling *register allocation*[122]. What it meanings for us, emitted code will contain less local variables.

For example, function calculating first S-box of DES encryption algorithm, it processing 32/64/128/256 values at once (depending on `DES_type` type (uint32, uint64, SSE2 or AVX)) using bitslice DES method (read more about this method here 1.20):

```
/*
 * 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,
```

[122] assigning variables to registers

```
    DES_type    *out4
) {
    DES_type    x1, x2, x3, x4, x5, x6, x7, x8;
    DES_type    x9, x10, x11, x12, x13, x14, x15, x16;
    DES_type    x17, x18, x19, x20, x21, x22, x23, x24;
    DES_type    x25, x26, x27, x28, x29, x30, x31, x32;
    DES_type    x33, x34, x35, x36, x37, x38, x39, x40;
    DES_type    x41, x42, x43, x44, x45, x46, x47, x48;
    DES_type    x49, x50, x51, x52, x53, x54, x55, x56;

    x1 = a3 & ~a5;
    x2 = x1 ^ a4;
    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;
}
```

There is a lot of local variables. Of course, not them all will be in local stack. Let's compile it with MSVC 2008 with /Ox option:

Listing 1.112: Optimizing MSVC 2008

```
PUBLIC    _s1
; Function compile flags: /Ogtpy
_TEXT    SEGMENT
_x6$ = -20          ; size = 4
_x3$ = -16          ; size = 4
_x1$ = -12          ; size = 4
_x8$ = -8           ; size = 4
_x4$ = -4           ; size = 4
_a1$ = 8            ; size = 4
_a2$ = 12           ; size = 4
_a3$ = 16           ; size = 4
_x33$ = 20          ; size = 4
_x7$ = 20           ; size = 4
_a4$ = 20           ; size = 4
_a5$ = 24           ; size = 4
tv326 = 28          ; size = 4
_x36$ = 28          ; size = 4
_x28$ = 28          ; size = 4
_a6$ = 28           ; size = 4
_out1$ = 32         ; size = 4
_x24$ = 36          ; size = 4
_out2$ = 36         ; size = 4
_out3$ = 40         ; size = 4
_out4$ = 44         ; size = 4
_s1    PROC
    sub    esp, 20                    ; 00000014H
    mov    edx, DWORD PTR _a5$[esp+16]
    push   ebx
    mov    ebx, DWORD PTR _a4$[esp+20]
    push   ebp
    push   esi
    mov    esi, DWORD PTR _a3$[esp+28]
    push   edi
    mov    edi, ebx
    not    edi
    mov    ebp, edi
    and    edi, DWORD PTR _a5$[esp+32]
    mov    ecx, edx
    not    ecx
    and    ebp, esi
    mov    eax, ecx
    and    eax, esi
    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 variables was allocated in local stack by compiler.

Now let's try the same thing in 64-bit version of MSVC 2008:

Listing 1.113: Optimizing 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
```

Nothing allocated in local stack by compiler, `x36` is synonym for `a5`.

By the way, we can see here, the function saved `RCX` and `RDX` registers in allocated by caller space, but `R8` and `R9` are not saved but used from the beginning.

By the way, there are CPUs with much more general purpose registers, Itanium, for example — 128 registers.

1.21.2 ARM

In ARM, 64-bit instructions are appeared in ARMv8.

1.22 C99 restrict

Here is a reason why FORTRAN programs, in some cases, works faster than C/C++ ones.

```
void f1 (int* x, int* y, int* sum, int* product, int* sum_product, int* update_me, size_t s)
{
        for (int i=0; i<s; i++)
        {
                sum[i]=x[i]+y[i];
                product[i]=x[i]*y[i];
                update_me[i]=i*123; // some dummy value
                sum_product[i]=sum[i]+product[i];
        };
};
```

That's very simple example with one specific thing in it: pointer to `update_me` array could be a pointer to `sum` array, `product` array, or even `sum_product` array — because, there are nothing criminal in it, right?

Compiler is fully aware about it, so it generates a code with four stages in loop body:

- calculate next `sum[i]`
- calculate next `product[i]`
- calculate next `update_me[i]`
- calculate next `sum_product[i]` — on this stage, we need to load from memory already calculated `sum[i]` and `product[i]`

Is it possible to optimize the last stage? Because already calculated `sum[i]` and `product[i]` are not necessary to load from memory again, becuse we already calculated them. Yes, but compiler isn't sure that nothing was overwritten on 3rd stage! This is called "pointer aliasing", a situation, when compiler can't be sure that a memory to which some pointer is pointing, wasn't changed.

restrict in C99 standard [ISO07, 6.7.3.1] is a promise, given by programmer to compiler that function arguments marked by this keyword will always be pointing to different memory locations and never be crossed.

If to be more precise and describe this formally, *restrict* shows that only this pointer will be used to access an object, with which we are working via this pointer, and no other pointer will be used for it. It can be even said that some object will be accessed only via one single pointer, if it's marked as *restrict*.

Let's add this keyword to each argument-pointer:

```
void f2 (int* restrict x, int* restrict y, int* restrict sum, int* restrict product, int* restrict
    sum_product,
        int* restrict update_me, size_t s)
{
        for (int i=0; i<s; i++)
        {
                sum[i]=x[i]+y[i];
                product[i]=x[i]*y[i];
                update_me[i]=i*123; // some dummy value
                sum_product[i]=sum[i]+product[i];
        };
};
```

Let's see results:

Listing 1.114: GCC x64: f1()

```
f1:
        push    r15 r14 r13 r12 rbp rdi rsi rbx
        mov     r13, QWORD PTR 120[rsp]
        mov     rbp, QWORD PTR 104[rsp]
        mov     r12, QWORD PTR 112[rsp]
        test    r13, r13
        je      .L1
        add     r13, 1
```

```
        xor     ebx, ebx
        mov     edi, 1
        xor     r11d, r11d
        jmp     .L4
.L6:
        mov     r11, rdi
        mov     rdi, rax
.L4:
        lea     rax, 0[0+r11*4]
        lea     r10, [rcx+rax]
        lea     r14, [rdx+rax]
        lea     rsi, [r8+rax]
        add     rax, r9
        mov     r15d, DWORD PTR [r10]
        add     r15d, DWORD PTR [r14]
        mov     DWORD PTR [rsi], r15d         ; store to sum[]
        mov     r10d, DWORD PTR [r10]
        imul    r10d, DWORD PTR [r14]
        mov     DWORD PTR [rax], r10d         ; store to product[]
        mov     DWORD PTR [r12+r11*4], ebx    ; store to update_me[]
        add     ebx, 123
        mov     r10d, DWORD PTR [rsi]         ; reload sum[i]
        add     r10d, DWORD PTR [rax]         ; reload product[i]
        lea     rax, 1[rdi]
        cmp     rax, r13
        mov     DWORD PTR 0[rbp+r11*4], r10d  ; store to sum_product[]
        jne     .L6
.L1:
        pop     rbx rsi rdi rbp r12 r13 r14 r15
        ret
```

Listing 1.115: GCC x64: f2()

```
f2:
        push    r13 r12 rbp rdi rsi rbx
        mov     r13, QWORD PTR 104[rsp]
        mov     rbp, QWORD PTR 88[rsp]
        mov     r12, QWORD PTR 96[rsp]
        test    r13, r13
        je      .L7
        add     r13, 1
        xor     r10d, r10d
        mov     edi, 1
        xor     eax, eax
        jmp     .L10
.L11:
        mov     rax, rdi
        mov     rdi, r11
.L10:
        mov     esi, DWORD PTR [rcx+rax*4]
        mov     r11d, DWORD PTR [rdx+rax*4]
        mov     DWORD PTR [r12+rax*4], r10d  ; store to update_me[]
        add     r10d, 123
        lea     ebx, [rsi+r11]
        imul    r11d, esi
        mov     DWORD PTR [r8+rax*4], ebx    ; store to sum[]
        mov     DWORD PTR [r9+rax*4], r11d   ; store to product[]
        add     r11d, ebx
        mov     DWORD PTR 0[rbp+rax*4], r11d ; store to sum_product[]
        lea     r11, 1[rdi]
        cmp     r11, r13
        jne     .L11
.L7:
        pop     rbx rsi rdi rbp r12 r13
        ret
```

The difference between compiled `f1()` and `f2()` function as follows: in `f1()`, `sum[i]` and `product[i]` are reloaded in the middle of loop, and in `f2()` there are no such thing, already calculated values are used, because we "promised" to compiler, that no one and nothing will change values in `sum[i]` and `product[i]` while execution of loop body, so it is "sure" that value from memory may not be again. Obviously, second example will work faster.

But what if pointers in function arguments will be crossed somehow? This will be on programmer's conscience, but results will be incorrect.

Let's back to FORTRAN. Compilers from this programming language treats all pointers as such, so when there are was not possible to set *restrict*, FORTRAN in these cases may generate faster code.

How practical is it? In the cases when function works with several big blocks in memory. There are a lot of such in linear algebra, for example. A lot of linear algebra used on supercomputers/HPC, probably, that's why, traditionally, FORTRAN is still used there [Loh10].

But when there are not a big number of iterations, certainly, speed boost will not be significant.

Chapter 2

Couple things to add

2.1 LEA instruction

LEA (*Load Effective Address*) is instruction intended not for values summing but for address forming, e.g., for forming address of array element by adding array address, element index, with multiplication of element size[1].

Important property of LEA instruction is that it do not alter processor flags.

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

MSVC 2010 with /Ox option:

```
_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] See also: http://en.wikipedia.org/wiki/Addressing_mode

2.2 Function prologue and epilogue

Function prologue is instructions at function start. It is often something like the following code fragment:

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

What these instruction do: save EBP register value, set EBP to ESP and then allocate space in stack for local variables.

EBP value is fixed over a period of function execution and it will be used for local variables and arguments access. One can use ESP, but it changing over time and it is not convenient.

Function epilogue annuled allocated space in stack, returning EBP value to initial state and returning control flow to callee:

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

Epilogue and prologue can make recursion performance worse.

For example, once upon a time I wrote a function to seek right node in binary tree. As a recursive function it would look stylish but because some time was spent at each function call for prologue/epilogue, it was working couple of times slower than the implementation without recursion.

By the way, that is the reason of tail call[2] existence: when compiler (or interpreter) transforms recursion (with which it's possible: *tail recursion*) into iteration for efficiency.

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

2.3 npad

It's an assembly language macro for label aligning by some specific border.

That's often need for the busy labels to where control flow is often passed, e.g., loop body begin. So the CPU will effectively load data or code from the memory, through memory bus, cache lines, etc.

Taken from `listing.inc` (MSVC):

By the way, it's curious example of different `NOP` variations. All these instructions has no effects whatsoever, but has different size.

```
;; 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
endm
```

2.4 Signed number representations

There are several methods of representing signed numbers[3], but in x86 architecture used "two's complement".

Difference between signed and unsigned numbers is that if we represent $0xFFFFFFFE$ and $0x0000002$ as unsigned, then first number (4294967294) is bigger than second (2). If to represent them both as signed, first will be -2, and it is lesser than second (2). That is the reason why conditional jumps 1.8 are present both for signed (e.g. `JG`, `JL`) and unsigned (`JA`, `JBE`) operations.

2.4.1 Integer overflow

It is worth noting that incorrect representation of number can lead integer overflow vulnerability.

For example, we have some network service, it receives network packets. In that packets there are also field where subpacket length is coded. It is 32-bit value. After network packet received, service checking that field, and if it is larger than, for example, some `MAX_PACKET_SIZE` (let's say, 10 kilobytes), packet ignored as incorrect. Comparison is signed. Intruder set this value to $0xFFFFFFFF$. While comparison, this number is considered as signed -1 and it's lesser than 10 kilobytes. No error here. Service would like to copy that subpacket to another place in memory and call `memcpy (dst, src, 0xFFFFFFFF)` function: this operation, rapidly scratching a lot of inside of process memory.

More about it: http://www.phrack.org/issues.html?issue=60&id=10

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

2.5 Arguments passing methods (calling conventions)

2.5.1 cdecl

This is the most popular method for arguments passing to functions in C/C++ languages.

Caller pushing arguments to stack in reverse order: last argument, then penultimate element and finally — first argument. Caller also should return back ESP to its initial state after callee function exit.

Listing 2.1: cdecl

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

2.5.2 stdcall

Almost the same thing as *cdecl*, with the exception that callee set ESP to initial state executing RET x instruction instead of RET, where x = arguments number * sizeof(int)[4]. Caller will not adjust stack pointer by add esp, x instruction.

Listing 2.2: stdcall

```
push arg3
push arg2
push arg1
call function

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

This method is ubiquitous in win32 standard libraries, but not in win64 (see below about win64).

Variable arguments number functions

printf()-like functions are, probably, the only case of variable arguments functions in C/C++, but it's easy to illustrate an important difference between *cdecl* and *stdcall* with help of it. Let's start with the idea that compiler knows argument count of each printf() function calling. However, called printf(), which is already compiled and located in MSVCRT.DLL (if to talk about Windows), has not any information about how much arguments were passed, however it can determine it from format string. Thus, if printf() would be *stdcall*-function and restored stack pointer to its initial state by counting number of arguments in format string, this could be dangerous situation, when one programmer's typo may provoke sudden program crash. Thus it's not suitable for such functions to use *stdcall*, *cdecl* is better.

2.5.3 fastcall

That's general naming for a method for passing some of arguments via registers and all others — via stack. It worked faster than *cdecl*/*stdcall* on older CPUs. It's not a standardized way, so, various compilers may do it differently. Well known caveat: if you have two DLLs, one uses another, and they are built by different compilers with different *fastcall* calling conventions.

Both MSVC and GCC passing first and second argument via ECX and EDX and other arguments via stack. Caller should restore stack pointer into initial state.

Stack pointer should be restored to initial state by callee, like in *stdcall*.

[4] Size of *int* type variable is 4 in x86 systems and 8 in x64 systems

Listing 2.3: fastcall

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

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

GCC regparm

It's *fastcall* evolution[5] is some sense. With the `-mregparm` option it's possible to set, how many arguments will be passed via registers. 3 at maximum. Thus, `EAX`, `EDX` and `ECX` registers will be used.

Of course, if number of arguments is less then 3, not all registers 3 will be used.

Caller restores stack pointer to its initial state.

2.5.4 thiscall

In C++, it's a *this* pointer to object passing into function-method.

In MSVC, *this* is usually passed in `ECX` register.

In GCC, *this* pointer is passed as a first function-method argument. Thus it will be seen that internally all function-methods has extra argument for it.

2.5.5 x86-64

win64

The method of arguments passing in Win64 is somewhat resembling to `fastcall`. First 4 arguments are passed via `RCX`, `RDX`, `R8`, `R9`, others — via stack. Caller also must prepare a place for 32 bytes or 4 64-bit values, so then callee can save there first 4 arguments. Short functions may use argument values just from registers, but larger may save its values for further use.

Caller also should return stack pointer into initial state.

This calling convention is also used in Windows x86-64 system DLLs (instead if *stdcall* in win32).

2.5.6 Returning values of *float* and *double* type

In all conventions except of Win64, values of type *float* or *double* returning via FPU register `ST(0)`.

In Win64, return values of *float* and *double* types are returned in `XMM0` register instead of `ST(0)`.

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

2.6 position-independent code

While analyzing Linux shared (.so) libraries, one may frequently spot such code pattern:

Listing 2.4: libc-2.17.so x86

```
.text:0012D5E3 __x86_get_pc_thunk_bx proc near         ; CODE XREF: sub_17350+3
.text:0012D5E3                                         ; sub_173CC+4 ...
.text:0012D5E3                 mov     ebx, [esp+0]
.text:0012D5E6                 retn
.text:0012D5E6 __x86_get_pc_thunk_bx endp

...

.text:000576C0 sub_576C0       proc near               ; CODE XREF: tmpfile+73

...

.text:000576C0                 push    ebp
.text:000576C1                 mov     ecx, large gs:0
.text:000576C8                 push    edi
.text:000576C9                 push    esi
.text:000576CA                 push    ebx
.text:000576CB                 call    __x86_get_pc_thunk_bx
.text:000576D0                 add     ebx, 157930h
.text:000576D6                 sub     esp, 9Ch

...

.text:000579F0                 lea     eax, (a__gen_tempname - 1AF000h)[ebx] ; "__gen_tempname"
.text:000579F6                 mov     [esp+0ACh+var_A0], eax
.text:000579FA                 lea     eax, (a__SysdepsPosix - 1AF000h)[ebx] ; "../sysdeps/posix/tempname.c"
.text:00057A00                 mov     [esp+0ACh+var_A8], eax
.text:00057A04                 lea     eax, (aInvalidKindIn_ - 1AF000h)[ebx] ; "! \"invalid KIND in
    __gen_tempname\""
.text:00057A0A                 mov     [esp+0ACh+var_A4], 14Ah
.text:00057A12                 mov     [esp+0ACh+var_AC], eax
.text:00057A15                 call    __assert_fail
```

All pointers to strings are corrected by some constant and by EBX value, which calculated at the beginning of each function. This is so called position-independent code (PIC), it is intended to execute placed in any random place of memory, that's why it can't contain any absolute memory addresses.

PIC was crucial in early computer systems and crucial now in embedded systems without virtual memory support (where processes are all placed in single continous memory block). It is also still used in *NIX systems for shared libraries, because, shared libraries are shared across many processes while loaded in memory only once. But all these processes may map the same shared library on different addresses, so that's why shared library should be working correctly without fixing on any absolute address.

Let's do some simple experiment:

```
#include <stdio.h>

int global_variable=123;

int f1(int var)
{
    int rt=global_variable+var;
    printf ("returning %d\n", rt);
    return rt;
};
```

Let's compile it in GCC 4.7.3 and see resulting .so file in IDA 5:

```
gcc -fPIC -shared -O3 -o 1.so 1.c
```

Listing 2.5: GCC 4.7.3

```
.text:00000440                 public __x86_get_pc_thunk_bx
.text:00000440 __x86_get_pc_thunk_bx proc near         ; CODE XREF: _init_proc+4
.text:00000440                                         ; deregister_tm_clones+4 ...
.text:00000440                 mov     ebx, [esp+0]
.text:00000443                 retn
.text:00000443 __x86_get_pc_thunk_bx endp

.text:00000570                 public f1
.text:00000570 f1              proc near
.text:00000570
.text:00000570 var_1C          = dword ptr -1Ch
.text:00000570 var_18          = dword ptr -18h
.text:00000570 var_14          = dword ptr -14h
.text:00000570 var_8           = dword ptr -8
.text:00000570 var_4           = dword ptr -4
.text:00000570 arg_0           = dword ptr  4
.text:00000570
.text:00000570                 sub     esp, 1Ch
.text:00000573                 mov     [esp+1Ch+var_8], ebx
.text:00000577                 call    __x86_get_pc_thunk_bx
.text:0000057C                 add     ebx, 1A84h
.text:00000582                 mov     [esp+1Ch+var_4], esi
.text:00000586                 mov     eax, ds:(global_variable_ptr - 2000h)[ebx]
.text:0000058C                 mov     esi, [eax]
.text:0000058E                 lea     eax, (aReturningD - 2000h)[ebx] ; "returning %d\n"
.text:00000594                 add     esi, [esp+1Ch+arg_0]
.text:00000598                 mov     [esp+1Ch+var_18], eax
.text:0000059C                 mov     [esp+1Ch+var_1C], 1
.text:000005A3                 mov     [esp+1Ch+var_14], esi
.text:000005A7                 call    ___printf_chk
.text:000005AC                 mov     eax, esi
.text:000005AE                 mov     ebx, [esp+1Ch+var_8]
.text:000005B2                 mov     esi, [esp+1Ch+var_4]
.text:000005B6                 add     esp, 1Ch
.text:000005B9                 retn
.text:000005B9 f1              endp
```

That's it: pointers to *«returning %d\n»* string and *global_variable* are to be corrected at each function execution. The *__x86_get_pc_thunk_bx()* function return address of the point after call to itself (0x57C here) in EBX. That's the simple way to get value of program counter (EIP) at some place. The *0x1A84* constant is related to the difference between this function begin and so called *Global Offset Table Procedure Linkage Table* (GOT PLT), the section right after *Global Offset Table* (GOT), where pointer to *global_variable* is. IDA 5 shows these offset processed, so to understand them easily, but in fact the code is:

```
.text:00000577                 call    __x86_get_pc_thunk_bx
.text:0000057C                 add     ebx, 1A84h
.text:00000582                 mov     [esp+1Ch+var_4], esi
.text:00000586                 mov     eax, [ebx-0Ch]
.text:0000058C                 mov     esi, [eax]
.text:0000058E                 lea     eax, [ebx-1A30h]
```

So, EBX pointing to GOT PLT section and to calculate pointer to *global_variable* which stored in GOT, 0xC should be subtracted. To calculate pointer to the *«returning %d\n»* string, 0x1A30 should be subtracted.

By the way, that is the reason why AMD64 instruction set supports RIP[6]-relative addressing, just to simplify PIC-code.

Let's compile the same C code in the same GCC version, but for x64.

IDA 5 would simplify output code but suppressing RIP-relative addressing details, so I'll run *objdump* instead to see the details:

```
0000000000000720 <f1>:
 720:   48 8b 05 b9 08 20 00    mov    rax,QWORD PTR [rip+0x2008b9]        # 200fe0 <_DYNAMIC+0x1d0>
```

[6] program counter in AMD64

```
727:   53                      push   rbx
728:   89 fb                   mov    ebx,edi
72a:   48 8d 35 20 00 00 00    lea    rsi,[rip+0x20]        # 751 <_fini+0x9>
731:   bf 01 00 00 00          mov    edi,0x1
736:   03 18                   add    ebx,DWORD PTR [rax]
738:   31 c0                   xor    eax,eax
73a:   89 da                   mov    edx,ebx
73c:   e8 df fe ff ff          call   620 <__printf_chk@plt>
741:   89 d8                   mov    eax,ebx
743:   5b                      pop    rbx
744:   c3                      ret
```

0x2008b9 is the difference between address of instruction at 0x720 and *global_variable* and 0x20 is the difference between that address of instruction at 0x72A and *«returning %d\n»* string.

The PIC mechanism is not used in Windows DLLs. If Windows loader needs to load DLL to another place, it "patches" DLL in memory (at the *FIXUP* places) in order to correct all addresses. This means, several Windows processes can't share once loaded DLL on different addresses in different process' memory blocks — because each loaded into memory DLL instance *fixed* to be work only at these addresses..

Chapter 3

Finding important/interesting stuff in the code

Minimalism it's not a significant feature of modern software.

But not because programmers writting a lot, but because all libraries are usually linked statically to executable files. If all external libraries were shifted into external DLL files, the world would be different. (Another reason for C++ — STL and other template libraries.)

Thus, it's very important to determine origin of some function, if it's from standard library or well-known library (like Boost[1], libpng[2]), and which one — is related to what we are trying to find in the code.

It's just absurdly to rewrite all code to C/C++ to find what we looking for.

One of the primary reverse engineer's task is to find quickly in the code what is needed.

IDA 5 disassembler allow us search among text strings, byte sequences, constants. It's even possible to export the code into .lst or .asm text file and then use `grep`, `awk`, etc.

When you try to understand what some code is doing, this easily could be some open-source library like libpng. So when you see some constants or text strings looks familiar, it's always worth to google it. And if you find the opensource project where it's used, then it will be enough just to compare the functions. It may solve some part of problem.

For example, once upon a time I tried to understand how SAP 6.0 network packets compression/decompression is working. It's a huge software, but a detailed .PDB with debugging information is present, and that's cosily. I finally came to idea that one of the functions doing decompressing of network packet called CsDecomprLZC(). Immediately I tried to google its name and I quickly found that the function named as the same is used in MaxDB (it's open-source SAP project)[3].

http://www.google.com/search?q=CsDecomprLZC

Astoundingly, MaxDB and SAP 6.0 software shared the same code for network packets compression/decompression.

3.1 Communication with the outer world

First what to look on is which functions from operation system API and standard libraries are used.

If the program is divided into main executable file and a group of DLL-files, sometimes, these function's names may be helpful.

If we are interesting, what exactly may lead to `MessageBox()` call with specific text, first what we can try to do: find this text in data segment, find references to it and find the points from which a control may be passed to `MessageBox()` call we're interesting in.

If we are talking about some game and we're interesting, which events are more or less random in it, we may try to find `rand()` function or its replacement (like Mersenne twister algorithm) and find a places from which this function called and most important: how the results are used.

But if it's not a game, but `rand()` is used, it's also interesing, why. There are cases of unexpected `rand()` usage in data compression algorithm (for encryption imitation): http://blog.yurichev.com/node/44.

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

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

[3] More about it in releval section 7.2.1

3.2 String

Debugging messages are often very helpful if present. In some sense, debugging messages are reporting about what's going on in program right now. Often these are `printf()`-like functions, which writes to log-files, and sometimes, not writing anything but calls are still present, because this build is not debug build but release one. If local or global variables are dumped in debugging messages, it might be helpful as well because it's possible to get variable names at least. For example, one of such functions in Oracle RDBMS is `ksdwrt()`.

Sometimes `assert()` macro presence is useful too: usually, this macro leave in code source file name, line number and condition.

Meaningful text strings are often helpful. IDA 5 disassembler may show from which function and from which point this specific string is used. Funny cases sometimes happen.

Paradoxically, but error messages may help us as well. In Oracle RDBMS, errors are reporting using group of functions. More about it.

It's possible to find very quickly, which functions reporting about errors and in which conditions. By the way, it's often a reason why copy-protection systems has inarticulate cryptic error messages or just error numbers. No one happy when software cracker quickly understand why copy-protection is triggered just by error message.

3.3 Constants

Some algorithms, especially cryptographical, use distinct constants, which is easy to find in code using IDA 5.

For example, MD5[4] algorithm initializes its own internal variables like:

```
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476
```

If you find these four constants usage in the code in a row — it's very high probability this function is related to MD5.

3.3.1 Magic numbers

A lot of file formats defining a standard file header where *magic number*[5] is used.

For example, all Win32 and MS-DOS executables are started with two characters "MZ"[6].

At the MIDI-file beginning "MThd" signature must be present. If we have a program that using MIDI-files for something, very likely, it will check MIDI-files for validity by checking at least first 4 bytes.

This could be done like:

(*buf* pointing to the beginning of loaded file into memory)

```
cmp [buf], 0x6468544D ; "MThd"
jnz _error_not_a_MIDI_file
```

...or by calling function for comparing memory blocks `memcmp()` or any other equivalent code up to `CMPSB` instruction.

When you find such place you already may say where MIDI-file loading is beginning, also, we could see a location of MIDI-file contents buffer and what is used from that buffer and how.

[4] http://en.wikipedia.org/wiki/MD5
[5] http://en.wikipedia.org/wiki/Magic_number_(programming)
[6] http://en.wikipedia.org/wiki/DOS_MZ_executable

DHCP

This applies to network protocols as well. For example, DHCP protocol network packets contains so called *magic cookie*: 0x63538263. Any code generating DHCP protocol packets somewhere and somehow should embed this constant into packet. If we find it in the code we may find where it happen and not only this. *Something* that received DHCP packet should check *magic cookie*, comparing it with the constant.

For example, let's take dhcpcore.dll file from Windows 7 x64 and search for the constant. And we found it, two times: it seems, that constant is used in two functions eloquently named as `DhcpExtractOptionsForValidation()` and `DhcpExtractFullOptions()`:

Listing 3.1: dhcpcore.dll (Windows 7 x64)

```
.rdata:000007FF6483CBE8 dword_7FF6483CBE8 dd 63538263h          ; DATA XREF: DhcpExtractOptionsForValidation
    +79
.rdata:000007FF6483CBEC dword_7FF6483CBEC dd 63538263h          ; DATA XREF: DhcpExtractFullOptions+97
```

And the places where these constants accessed:

Listing 3.2: dhcpcore.dll (Windows 7 x64)

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

And:

Listing 3.3: dhcpcore.dll (Windows 7 x64)

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

3.4 Finding the right instructions

If the program is using FPU instructions and there are very few of them in a code, one can try to check each by debugger.

For example, we may be interesting, how Microsoft Excel calculating formulae entered by user. For example, division operation.

If to load excel.exe (from Office 2010) version 14.0.4756.1000 into IDA 5, then make a full listing and to find each `FDIV` instructions (except ones which use constants as a second operand — obviously, it's not suits us):

```
cat EXCEL.lst | grep fdiv | grep -v dbl_ > EXCEL.fdiv
```

...then we realizing they are just 144.

We can enter string like `=(1/3)` in Excel and check each instruction.

Checking each instruction in debugger or *tracer* 5.0.1 (one may check 4 instruction at a time), it seems, we are lucky here and sought-for instruction is just 14th:

```
.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
```

`ST(0)` holding first argument (1) and second one is in `[ebx]`.

Next instruction after `FDIV` writes result into memory:

```
.text:3011E91B DD 1E                                fstp    qword ptr [esi]
```

If to set breakpoint on it, we may see result:

```
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
```

Also, as a practical joke, we can modify it on-fly:

```
tracer -l:excel.exe bpx=excel.exe!BASE+0x11E91B,set(st0,666)
```

```
PID=36540|TID=24056|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00680006 EBX=0x00680018 ECX=0x00000001 EDX=0x00000001
ESI=0x00680000 EDI=0x00395404 EBP=0x0290FD9C ESP=0x0290FD58
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
Set ST0 register to 666.000000
```

Excel showing 666 in that cell what finally convincing us we find the right place.

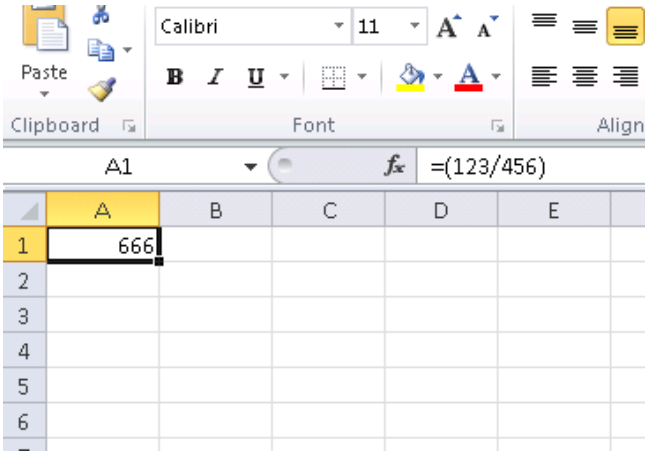

Figure 3.1: Practical joke worked

If to try the same Excel version, but x64, we'll see there are only 12 `FDIV` instructions, and the one we looking for — third.

```
tracer.exe -l:excel.exe bpx=excel.exe!BASE+0x1B7FCC,set(st0,666)
```

It seems, a lot of division operations of *float* and *double* types, compiler replaced by SSE-instructions like `DIVSD` (`DIVSD` present here 268 in total).

3.5 Suspicious code patterns

Modern compilers do not emit `LOOP` and `RCL` instructions. On the other hand, these instructions are well-known to coders who like to code in straight assembly language. If you spot these, it can be said, with a high probability, this fragment of code is hand-written. Also, function prologue/epilogue is not usually present in hand-written assembly copy.

3.6 Magic numbers usage while tracing

Often, main goal is to get to know, how some value was read from file, or received via network, being used. Often, manual tracing of some value is very labouring task. One of the simple methods (however, not 100% reliable) is to use your own *magic number*.

This resembling X-ray computed tomography is some sense: radiocontrast agent is often injected into patient's blood, which is used for improving visibility of internal structures in X-rays. For example, it's well known how blood of healthy man/woman percolates in kidneys and if agent is in blood, it will be easily seen on tomography, how good and normal blood was percolating, are there any stones or tumors.

We can take some 32-bit number like *0x0badf00d*, or someone birth date like *0x11101979* and to write this, 4 byte holding number, to some place in file used by the program we investigate.

Then, while tracing this program, with *tracer* 5.0.1 in *code coverage* mode, and then, with help if *grep* or just by searching in the text file (of tracing results), we can easily see, where the value was used and how.

Example of *grepable tracer* 5.0.1 results in *cc* mode:

```
0x150bf66 (_kziaia+0x14), e=       1 [MOV EBX, [EBP+8]] [EBP+8]=0xf59c934
0x150bf69 (_kziaia+0x17), e=       1 [MOV EDX, [69AEB08h]] [69AEB08h]=0
0x150bf6f (_kziaia+0x1d), e=       1 [FS: MOV EAX, [2Ch]]
0x150bf75 (_kziaia+0x23), e=       1 [MOV ECX, [EAX+EDX*4]] [EAX+EDX*4]=0xf1ac360
0x150bf78 (_kziaia+0x26), e=       1 [MOV [EBP-4], ECX] ECX=0xf1ac360
```

This can be used for network packets as well. It's important to be unique for *magic number* and not to be present in the program's code.

Aside of *tracer* 5.0.1, DosBox (MS-DOS emulator) in heavydebug mode, can write information about all register's states for each executed instruction of program to plain text file[7], so this method may be useful for DOS programs as well.

3.7 Old-school methods, nevertheless, interesting to know

3.7.1 Memory "snapshots" comparing

The method of simple two memory snapshots comparing in order to see changes, was often used to hack 8-bit computer games and hacking "high score" files.

For example, if you got some loaded game on 8-bit computer (it's not much memory on these, but game is usually consumes even less memory) and you know that you have now, let's say, 100 bullets, you can do a "snapshot" of all memory and save it to some place. Then shoot somewhere, bullet count now 99, do second "snapshot" and then compare both: somewhere should be a byte which was 100 in the beginning and now it's 99. Considering a fact that these 8-bit games were often written in assembly language and such variables were global, it can be said for sure, which address in memory holding bullets count. If to search all references to that address in disassembled game code, it's not very hard to find a piece of code decrementing bullets count, write `NOP` instruction[8] there, or couple of `NOP`-s, we'll have a game with 100 (for example) bullets forever. Games on these 8-bit computers was usually loaded on the same address, also, there were no much different versions of each game (usually, just one version is popular), enthusiastic gamers knew, which byte should be written (using BASIC instruction *POKE*[9]) to which address in order to hack it. This led to "cheat" lists containing of *POKE* instructions published in magazines related to 8-bit games. See also: http://en.wikipedia.org/wiki/PEEK_and_POKE.

The same story about modifying "high score" files, this may work not only with 8-bit games. Let's notice your score count and save the file somewhere. When "high score" count will be different, just compare two files, it can be even done with DOS-utility FC[10] ("high score" files are often in binary form). There will be some place where couple of bytes will be different and it will be easy to see which ones are holding score number. However, game developers are aware of such tricks and may protect against it.

[7] See also my blog post about this DosBox feature: http://blog.yurichev.com/node/55

[8] "no operation", idle operation

[9] BASIC language instruction writting byte on specific address

[10] MS-DOS utility for binary files comparing

Chapter 4

Tasks

There are two questions almost for every task, if otherwise isn't specified:
1) What this function does? Answer in one-sentence form.
2) Rewrite this function into C/C++.
Hints and solutions are in the appendix of this book.

4.1 Easy level

4.1.1 Task 1.1

This is standard C library function. Source code taken from OpenWatcom. Compiled in 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
```

It is the same code compiled by GCC 4.4.1 with -O3 option (maximum optimization):

```
_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
```

4.1.2 Task 1.2

This is also standard C library function. Source code is taken from OpenWatcom and modified slightly. Compiled in MSVC 2010 with /Ox optimization flag.

This function also use these standard C functions: isspace() and 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
```

Same code compiled in GCC 4.4.1. This task is sligthly harder because GCC compiled isspace() and isdigit() functions like inline-functions and inserted their bodies right into code.

```
_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
```

4.1.3 Task 1.3

This is standard C function too, actually, two functions working in pair. Source code taken from MSVC 2010 and modified sligthly.

The matter of modification is that this function can work properly in multi-threaded environment, and I removed its support for simplification (or for confusion).

Compiled in MSVC 2010 with `/Ox` flag.

```
_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
```

Same code compiled in 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
```

4.1.4 Task 1.4

This is standard C library function. Source code taken from MSVC 2010. Compiled in MSVC 2010 with `/Ox` flag.

```
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
```

Same code compiled in 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
```

4.1.5 Task 1.5

This task is rather on knowledge than on reading code.

The function is taken from OpenWatcom. Compiled in MSVC 2010 with `/Ox` flag.

```
_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
```

4.1.6 Task 1.6

Compiled in MSVC 2010 with `/Ox` option.

```
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
```

4.1.7 Task 1.7

This function is taken from Linux 2.6 kernel.

Compiled in MSVC 2010 with `/Ox` option:

```
_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
```

4.1.8 Task 1.8

Compiled in MSVC 2010 with /O1 option[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
```

4.1.9 Task 1.9

Compiled in MSVC 2010 with /O1 option:

```
tv315 = -8              ; size = 4
tv291 = -4              ; size = 4
_a$ = 8                 ; size = 4
_b$ = 12                ; size = 4
_c$ = 16                ; size = 4
?m@@YAXPAN00@Z PROC     ; m, COMDAT
```

[1] /O1: minimize space

```
    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
```

4.1.10 Task 1.10

If to compile this piece of code and run, some number will be printed. Where it came from? Where it came from if to compile it in MSVC with optimization (`/Ox`)?

```
#include <stdio.h>

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

        return 0;
};
```

4.2 Middle level

4.2.1 Task 2.1

Well-known algorithm, also included in standard C library. Source code was taken from glibc 2.11.1. Compiled in GCC 4.4.1 with `-Os` option (code size optimization). Listing was done by IDA 4.9 disassembler from ELF-file generated by GCC and linker.

For those who wants use IDA while learning, here you may find .elf and .idb files, .idb can be opened with freeware 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
```

4.3 crackme / keygenme

Couple of my keygenmes[2]:
http://crackmes.de/users/yonkie/

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

Chapter 5

Tools

- IDA as disassembler. Older freeware version is available for downloading: http://www.hex-rays.com/idapro/idadownfreeware.htm.
- Microsoft Visual Studio Express[1]: Stripped-down Visual Studio version, convenient for simple expreiments.
- Hiew[2] for small modifications of code in binary files.

5.0.1 Debugger

tracer[3] instead of debugger.

I stopped to use debugger eventually, because all I need from it is to spot some function's arguments while execution, or registers' state at some point. To load debugger each time is too much, so I wrote a small utility *tracer*. It has console-interface, working from command-line, enable us to intercept function execution, set breakpoints at arbitrary places, spot registers' state, modify it, etc.

However, as for learning, it's highly advisable to trace code in debugger manually, watch how register's state changing (for example, classic SoftICE, OllyDbg, WinDbg highlighting changed registers), flags, data, change them manually, watch reaction, etc.

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

Chapter 6

Books/blogs worth reading

6.1 Books

6.1.1 Windows

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

6.1.2 C/C++

- C++ language standard: ISO/IEC 14882:2003[2]

6.1.3 x86 / x86-64

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

6.1.4 ARM

ARM manuals: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html

6.2 Blogs

6.2.1 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

Chapter 7

More examples

7.1 “QR9”: Rubik’s cube inspired amateur crypto-algorithm

Sometimes amateur cryptosystems appear to be pretty bizarre.

I was asked to revese engineer an amateur cryptoalgorithm of some data crypting utility, source code of which was lost[1].

Here is also IDA 5 exported listing from original crypting utility:

```
.text:00541000 set_bit         proc near               ; CODE XREF: rotate1+42
.text:00541000                                         ; rotate2+42 ...
.text:00541000
.text:00541000 arg_0           = dword ptr  4
.text:00541000 arg_4           = dword ptr  8
.text:00541000 arg_8           = dword ptr  0Ch
.text:00541000 arg_C           = byte ptr  10h
.text:00541000
.text:00541000                 mov     al, [esp+arg_C]
.text:00541004                 mov     ecx, [esp+arg_8]
.text:00541008                 push    esi
.text:00541009                 mov     esi, [esp+4+arg_0]
.text:0054100D                 test    al, al
.text:0054100F                 mov     eax, [esp+4+arg_4]
.text:00541013                 mov     dl, 1
.text:00541015                 jz      short loc_54102B
.text:00541017                 shl     dl, cl
.text:00541019                 mov     cl, cube64[eax+esi*8]
.text:00541020                 or      cl, dl
.text:00541022                 mov     cube64[eax+esi*8], cl
.text:00541029                 pop     esi
.text:0054102A                 retn
.text:0054102B ; ---------------------------------------------------------------------------
.text:0054102B
.text:0054102B loc_54102B:                             ; CODE XREF: set_bit+15
.text:0054102B                 shl     dl, cl
.text:0054102D                 mov     cl, cube64[eax+esi*8]
.text:00541034                 not     dl
.text:00541036                 and     cl, dl
.text:00541038                 mov     cube64[eax+esi*8], cl
.text:0054103F                 pop     esi
.text:00541040                 retn
.text:00541040 set_bit         endp
.text:00541040
.text:00541040 ; ---------------------------------------------------------------------------
.text:00541041                 align 10h
.text:00541050
.text:00541050 ; =============== S U B R O U T I N E =======================================
.text:00541050
.text:00541050
```

[1] I also got permit from customer to publish the algorithm details

```
.text:00541050 get_bit         proc near               ; CODE XREF: rotate1+16
.text:00541050                                         ; rotate2+16 ...
.text:00541050
.text:00541050 arg_0           = dword ptr  4
.text:00541050 arg_4           = dword ptr  8
.text:00541050 arg_8           = byte ptr  0Ch
.text:00541050
.text:00541050                 mov     eax, [esp+arg_4]
.text:00541054                 mov     ecx, [esp+arg_0]
.text:00541058                 mov     al, cube64[eax+ecx*8]
.text:0054105F                 mov     cl, [esp+arg_8]
.text:00541063                 shr     al, cl
.text:00541065                 and     al, 1
.text:00541067                 retn
.text:00541067 get_bit         endp
.text:00541067
.text:00541067 ; ---------------------------------------------------------------------------
.text:00541068                 align 10h
.text:00541070
.text:00541070 ; =============== S U B R O U T I N E =======================================
.text:00541070
.text:00541070
.text:00541070 rotate1         proc near               ; CODE XREF: rotate_all_with_password+8E
.text:00541070
.text:00541070 internal_array_64= byte ptr -40h
.text:00541070 arg_0           = dword ptr  4
.text:00541070
.text:00541070                 sub     esp, 40h
.text:00541073                 push    ebx
.text:00541074                 push    ebp
.text:00541075                 mov     ebp, [esp+48h+arg_0]
.text:00541079                 push    esi
.text:0054107A                 push    edi
.text:0054107B                 xor     edi, edi        ; EDI is loop1 counter
.text:0054107D                 lea     ebx, [esp+50h+internal_array_64]
.text:00541081
.text:00541081 first_loop1_begin:                      ; CODE XREF: rotate1+2E
.text:00541081                 xor     esi, esi        ; ESI is loop2 counter
.text:00541083
.text:00541083 first_loop2_begin:                      ; CODE XREF: rotate1+25
.text:00541083                 push    ebp             ; arg_0
.text:00541084                 push    esi
.text:00541085                 push    edi
.text:00541086                 call    get_bit
.text:0054108B                 add     esp, 0Ch
.text:0054108E                 mov     [ebx+esi], al   ; store to internal array
.text:00541091                 inc     esi
.text:00541092                 cmp     esi, 8
.text:00541095                 jl      short first_loop2_begin
.text:00541097                 inc     edi
.text:00541098                 add     ebx, 8
.text:0054109B                 cmp     edi, 8
.text:0054109E                 jl      short first_loop1_begin
.text:005410A0                 lea     ebx, [esp+50h+internal_array_64]
.text:005410A4                 mov     edi, 7          ; EDI is loop1 counter, initial state is 7
.text:005410A9
.text:005410A9 second_loop1_begin:                     ; CODE XREF: rotate1+57
.text:005410A9                 xor     esi, esi        ; ESI is loop2 counter
.text:005410AB
.text:005410AB second_loop2_begin:                     ; CODE XREF: rotate1+4E
.text:005410AB                 mov     al, [ebx+esi]   ; value from internal array
.text:005410AE                 push    eax
.text:005410AF                 push    ebp             ; arg_0
.text:005410B0                 push    edi
.text:005410B1                 push    esi
.text:005410B2                 call    set_bit
```

```
.text:005410B7                 add     esp, 10h
.text:005410BA                 inc     esi             ; increment loop2 counter
.text:005410BB                 cmp     esi, 8
.text:005410BE                 jl      short second_loop2_begin
.text:005410C0                 dec     edi             ; decrement loop2 counter
.text:005410C1                 add     ebx, 8
.text:005410C4                 cmp     edi, 0FFFFFFFFh
.text:005410C7                 jg      short second_loop1_begin
.text:005410C9                 pop     edi
.text:005410CA                 pop     esi
.text:005410CB                 pop     ebp
.text:005410CC                 pop     ebx
.text:005410CD                 add     esp, 40h
.text:005410D0                 retn
.text:005410D0 rotate1         endp
.text:005410D0
.text:005410D0 ; ---------------------------------------------------------------------------
.text:005410D1                 align 10h
.text:005410E0
.text:005410E0 ; =============== S U B R O U T I N E =======================================
.text:005410E0
.text:005410E0
.text:005410E0 rotate2         proc near               ; CODE XREF: rotate_all_with_password+7A
.text:005410E0
.text:005410E0 internal_array_64= byte ptr -40h
.text:005410E0 arg_0           = dword ptr  4
.text:005410E0
.text:005410E0                 sub     esp, 40h
.text:005410E3                 push    ebx
.text:005410E4                 push    ebp
.text:005410E5                 mov     ebp, [esp+48h+arg_0]
.text:005410E9                 push    esi
.text:005410EA                 push    edi
.text:005410EB                 xor     edi, edi        ; loop1 counter
.text:005410ED                 lea     ebx, [esp+50h+internal_array_64]
.text:005410F1
.text:005410F1 loc_5410F1:                             ; CODE XREF: rotate2+2E
.text:005410F1                 xor     esi, esi        ; loop2 counter
.text:005410F3
.text:005410F3 loc_5410F3:                             ; CODE XREF: rotate2+25
.text:005410F3                 push    esi             ; loop2
.text:005410F4                 push    edi             ; loop1
.text:005410F5                 push    ebp             ; arg_0
.text:005410F6                 call    get_bit
.text:005410FB                 add     esp, 0Ch
.text:005410FE                 mov     [ebx+esi], al   ; store to internal array
.text:00541101                 inc     esi             ; increment loop1 counter
.text:00541102                 cmp     esi, 8
.text:00541105                 jl      short loc_5410F3
.text:00541107                 inc     edi             ; increment loop2 counter
.text:00541108                 add     ebx, 8
.text:0054110B                 cmp     edi, 8
.text:0054110E                 jl      short loc_5410F1
.text:00541110                 lea     ebx, [esp+50h+internal_array_64]
.text:00541114                 mov     edi, 7          ; loop1 counter is initial state 7
.text:00541119
.text:00541119 loc_541119:                             ; CODE XREF: rotate2+57
.text:00541119                 xor     esi, esi        ; loop2 counter
.text:0054111B
.text:0054111B loc_54111B:                             ; CODE XREF: rotate2+4E
.text:0054111B                 mov     al, [ebx+esi]   ; get byte from internal array
.text:0054111E                 push    eax
.text:0054111F                 push    edi             ; loop1 counter
.text:00541120                 push    esi             ; loop2 counter
.text:00541121                 push    ebp             ; arg_0
.text:00541122                 call    set_bit
```

```
.text:00541127                 add     esp, 10h
.text:0054112A                 inc     esi             ; increment loop2 counter
.text:0054112B                 cmp     esi, 8
.text:0054112E                 jl      short loc_54111B
.text:00541130                 dec     edi             ; decrement loop2 counter
.text:00541131                 add     ebx, 8
.text:00541134                 cmp     edi, 0FFFFFFFFh
.text:00541137                 jg      short loc_541119
.text:00541139                 pop     edi
.text:0054113A                 pop     esi
.text:0054113B                 pop     ebp
.text:0054113C                 pop     ebx
.text:0054113D                 add     esp, 40h
.text:00541140                 retn
.text:00541140 rotate2         endp
.text:00541140
.text:00541140 ; ---------------------------------------------------------------------------
.text:00541141                 align 10h
.text:00541150
.text:00541150 ; =============== S U B R O U T I N E =======================================
.text:00541150
.text:00541150
.text:00541150 rotate3         proc near               ; CODE XREF: rotate_all_with_password+66
.text:00541150
.text:00541150 var_40          = byte ptr -40h
.text:00541150 arg_0           = dword ptr  4
.text:00541150
.text:00541150                 sub     esp, 40h
.text:00541153                 push    ebx
.text:00541154                 push    ebp
.text:00541155                 mov     ebp, [esp+48h+arg_0]
.text:00541159                 push    esi
.text:0054115A                 push    edi
.text:0054115B                 xor     edi, edi
.text:0054115D                 lea     ebx, [esp+50h+var_40]
.text:00541161
.text:00541161 loc_541161:                             ; CODE XREF: rotate3+2E
.text:00541161                 xor     esi, esi
.text:00541163
.text:00541163 loc_541163:                             ; CODE XREF: rotate3+25
.text:00541163                 push    esi
.text:00541164                 push    ebp
.text:00541165                 push    edi
.text:00541166                 call    get_bit
.text:0054116B                 add     esp, 0Ch
.text:0054116E                 mov     [ebx+esi], al
.text:00541171                 inc     esi
.text:00541172                 cmp     esi, 8
.text:00541175                 jl      short loc_541163
.text:00541177                 inc     edi
.text:00541178                 add     ebx, 8
.text:0054117B                 cmp     edi, 8
.text:0054117E                 jl      short loc_541161
.text:00541180                 xor     ebx, ebx
.text:00541182                 lea     edi, [esp+50h+var_40]
.text:00541186
.text:00541186 loc_541186:                             ; CODE XREF: rotate3+54
.text:00541186                 mov     esi, 7
.text:0054118B
.text:0054118B loc_54118B:                             ; CODE XREF: rotate3+4E
.text:0054118B                 mov     al, [edi]
.text:0054118D                 push    eax
.text:0054118E                 push    ebx
.text:0054118F                 push    ebp
.text:00541190                 push    esi
.text:00541191                 call    set_bit
```

```
.text:00541196                 add     esp, 10h
.text:00541199                 inc     edi
.text:0054119A                 dec     esi
.text:0054119B                 cmp     esi, 0FFFFFFFFh
.text:0054119E                 jg      short loc_54118B
.text:005411A0                 inc     ebx
.text:005411A1                 cmp     ebx, 8
.text:005411A4                 jl      short loc_541186
.text:005411A6                 pop     edi
.text:005411A7                 pop     esi
.text:005411A8                 pop     ebp
.text:005411A9                 pop     ebx
.text:005411AA                 add     esp, 40h
.text:005411AD                 retn
.text:005411AD rotate3         endp
.text:005411AD
.text:005411AD ; ---------------------------------------------------------------------------
.text:005411AE                 align 10h
.text:005411B0
.text:005411B0 ; =============== S U B R O U T I N E =======================================
.text:005411B0
.text:005411B0
.text:005411B0 rotate_all_with_password proc near      ; CODE XREF: crypt+1F
.text:005411B0                                         ; decrypt+36
.text:005411B0
.text:005411B0 arg_0           = dword ptr  4
.text:005411B0 arg_4           = dword ptr  8
.text:005411B0
.text:005411B0                 mov     eax, [esp+arg_0]
.text:005411B4                 push    ebp
.text:005411B5                 mov     ebp, eax
.text:005411B7                 cmp     byte ptr [eax], 0
.text:005411BA                 jz      exit
.text:005411C0                 push    ebx
.text:005411C1                 mov     ebx, [esp+8+arg_4]
.text:005411C5                 push    esi
.text:005411C6                 push    edi
.text:005411C7
.text:005411C7 loop_begin:                             ; CODE XREF: rotate_all_with_password+9F
.text:005411C7                 movsx   eax, byte ptr [ebp+0]
.text:005411CB                 push    eax             ; C
.text:005411CC                 call    _tolower
.text:005411D1                 add     esp, 4
.text:005411D4                 cmp     al, 'a'
.text:005411D6                 jl      short next_character_in_password
.text:005411D8                 cmp     al, 'z'
.text:005411DA                 jg      short next_character_in_password
.text:005411DC                 movsx   ecx, al
.text:005411DF                 sub     ecx, 'a'
.text:005411E2                 cmp     ecx, 24
.text:005411E5                 jle     short skip_subtracting
.text:005411E7                 sub     ecx, 24
.text:005411EA
.text:005411EA skip_subtracting:                       ; CODE XREF: rotate_all_with_password+35
.text:005411EA                 mov     eax, 55555556h
.text:005411EF                 imul    ecx
.text:005411F1                 mov     eax, edx
.text:005411F3                 shr     eax, 1Fh
.text:005411F6                 add     edx, eax
.text:005411F8                 mov     eax, ecx
.text:005411FA                 mov     esi, edx
.text:005411FC                 mov     ecx, 3
.text:00541201                 cdq
.text:00541202                 idiv    ecx
.text:00541204                 sub     edx, 0
.text:00541207                 jz      short call_rotate1
```

```
.text:00541209                 dec     edx
.text:0054120A                 jz      short call_rotate2
.text:0054120C                 dec     edx
.text:0054120D                 jnz     short next_character_in_password
.text:0054120F                 test    ebx, ebx
.text:00541211                 jle     short next_character_in_password
.text:00541213                 mov     edi, ebx
.text:00541215
.text:00541215 call_rotate3:                           ; CODE XREF: rotate_all_with_password+6F
.text:00541215                 push    esi
.text:00541216                 call    rotate3
.text:0054121B                 add     esp, 4
.text:0054121E                 dec     edi
.text:0054121F                 jnz     short call_rotate3
.text:00541221                 jmp     short next_character_in_password
.text:00541223 ; ---------------------------------------------------------------------------
.text:00541223
.text:00541223 call_rotate2:                           ; CODE XREF: rotate_all_with_password+5A
.text:00541223                 test    ebx, ebx
.text:00541225                 jle     short next_character_in_password
.text:00541227                 mov     edi, ebx
.text:00541229
.text:00541229 loc_541229:                             ; CODE XREF: rotate_all_with_password+83
.text:00541229                 push    esi
.text:0054122A                 call    rotate2
.text:0054122F                 add     esp, 4
.text:00541232                 dec     edi
.text:00541233                 jnz     short loc_541229
.text:00541235                 jmp     short next_character_in_password
.text:00541237 ; ---------------------------------------------------------------------------
.text:00541237
.text:00541237 call_rotate1:                           ; CODE XREF: rotate_all_with_password+57
.text:00541237                 test    ebx, ebx
.text:00541239                 jle     short next_character_in_password
.text:0054123B                 mov     edi, ebx
.text:0054123D
.text:0054123D loc_54123D:                             ; CODE XREF: rotate_all_with_password+97
.text:0054123D                 push    esi
.text:0054123E                 call    rotate1
.text:00541243                 add     esp, 4
.text:00541246                 dec     edi
.text:00541247                 jnz     short loc_54123D
.text:00541249
.text:00541249 next_character_in_password:             ; CODE XREF: rotate_all_with_password+26
.text:00541249                                         ; rotate_all_with_password+2A ...
.text:00541249                 mov     al, [ebp+1]
.text:0054124C                 inc     ebp
.text:0054124D                 test    al, al
.text:0054124F                 jnz     loop_begin
.text:00541255                 pop     edi
.text:00541256                 pop     esi
.text:00541257                 pop     ebx
.text:00541258
.text:00541258 exit:                                   ; CODE XREF: rotate_all_with_password+A
.text:00541258                 pop     ebp
.text:00541259                 retn
.text:00541259 rotate_all_with_password endp
.text:00541259
.text:00541259 ; ---------------------------------------------------------------------------
.text:0054125A                 align 10h
.text:00541260
.text:00541260 ; =============== S U B R O U T I N E =======================================
.text:00541260
.text:00541260
.text:00541260 crypt           proc near               ; CODE XREF: crypt_file+8A
.text:00541260
```

```
.text:00541260 arg_0           = dword ptr  4
.text:00541260 arg_4           = dword ptr  8
.text:00541260 arg_8           = dword ptr  0Ch
.text:00541260
.text:00541260                 push    ebx
.text:00541261                 mov     ebx, [esp+4+arg_0]
.text:00541265                 push    ebp
.text:00541266                 push    esi
.text:00541267                 push    edi
.text:00541268                 xor     ebp, ebp
.text:0054126A
.text:0054126A loc_54126A:                             ; CODE XREF: crypt+41
.text:0054126A                 mov     eax, [esp+10h+arg_8]
.text:0054126E                 mov     ecx, 10h
.text:00541273                 mov     esi, ebx
.text:00541275                 mov     edi, offset cube64
.text:0054127A                 push    1
.text:0054127C                 push    eax
.text:0054127D                 rep movsd
.text:0054127F                 call    rotate_all_with_password
.text:00541284                 mov     eax, [esp+18h+arg_4]
.text:00541288                 mov     edi, ebx
.text:0054128A                 add     ebp, 40h
.text:0054128D                 add     esp, 8
.text:00541290                 mov     ecx, 10h
.text:00541295                 mov     esi, offset cube64
.text:0054129A                 add     ebx, 40h
.text:0054129D                 cmp     ebp, eax
.text:0054129F                 rep movsd
.text:005412A1                 jl      short loc_54126A
.text:005412A3                 pop     edi
.text:005412A4                 pop     esi
.text:005412A5                 pop     ebp
.text:005412A6                 pop     ebx
.text:005412A7                 retn
.text:005412A7 crypt           endp
.text:005412A7
.text:005412A7 ; ---------------------------------------------------------------------------
.text:005412A8                 align 10h
.text:005412B0
.text:005412B0 ; =============== S U B R O U T I N E =======================================
.text:005412B0
.text:005412B0
.text:005412B0 ; int __cdecl decrypt(int, int, void *Src)
.text:005412B0 decrypt         proc near               ; CODE XREF: decrypt_file+99
.text:005412B0
.text:005412B0 arg_0           = dword ptr  4
.text:005412B0 arg_4           = dword ptr  8
.text:005412B0 Src             = dword ptr  0Ch
.text:005412B0
.text:005412B0                 mov     eax, [esp+Src]
.text:005412B4                 push    ebx
.text:005412B5                 push    ebp
.text:005412B6                 push    esi
.text:005412B7                 push    edi
.text:005412B8                 push    eax             ; Src
.text:005412B9                 call    __strdup
.text:005412BE                 push    eax             ; Str
.text:005412BF                 mov     [esp+18h+Src], eax
.text:005412C3                 call    __strrev
.text:005412C8                 mov     ebx, [esp+18h+arg_0]
.text:005412CC                 add     esp, 8
.text:005412CF                 xor     ebp, ebp
.text:005412D1
.text:005412D1 loc_5412D1:                             ; CODE XREF: decrypt+58
.text:005412D1                 mov     ecx, 10h
```

```
.text:005412D6                 mov     esi, ebx
.text:005412D8                 mov     edi, offset cube64
.text:005412DD                 push    3
.text:005412DF                 rep movsd
.text:005412E1                 mov     ecx, [esp+14h+Src]
.text:005412E5                 push    ecx
.text:005412E6                 call    rotate_all_with_password
.text:005412EB                 mov     eax, [esp+18h+arg_4]
.text:005412EF                 mov     edi, ebx
.text:005412F1                 add     ebp, 40h
.text:005412F4                 add     esp, 8
.text:005412F7                 mov     ecx, 10h
.text:005412FC                 mov     esi, offset cube64
.text:00541301                 add     ebx, 40h
.text:00541304                 cmp     ebp, eax
.text:00541306                 rep movsd
.text:00541308                 jl      short loc_5412D1
.text:0054130A                 mov     edx, [esp+10h+Src]
.text:0054130E                 push    edx             ; Memory
.text:0054130F                 call    _free
.text:00541314                 add     esp, 4
.text:00541317                 pop     edi
.text:00541318                 pop     esi
.text:00541319                 pop     ebp
.text:0054131A                 pop     ebx
.text:0054131B                 retn
.text:0054131B decrypt         endp
.text:0054131B
.text:0054131B ; ---------------------------------------------------------------------------
.text:0054131C                 align 10h
.text:00541320
.text:00541320 ; =============== S U B R O U T I N E =======================================
.text:00541320
.text:00541320
.text:00541320 ; int __cdecl crypt_file(int Str, char *Filename, int password)
.text:00541320 crypt_file      proc near               ; CODE XREF: _main+42
.text:00541320
.text:00541320 Str             = dword ptr  4
.text:00541320 Filename        = dword ptr  8
.text:00541320 password        = dword ptr  0Ch
.text:00541320
.text:00541320                 mov     eax, [esp+Str]
.text:00541324                 push    ebp
.text:00541325                 push    offset Mode     ; "rb"
.text:0054132A                 push    eax             ; Filename
.text:0054132B                 call    _fopen          ; open file
.text:00541330                 mov     ebp, eax
.text:00541332                 add     esp, 8
.text:00541335                 test    ebp, ebp
.text:00541337                 jnz     short loc_541348
.text:00541339                 push    offset Format   ; "Cannot open input file!\n"
.text:0054133E                 call    _printf
.text:00541343                 add     esp, 4
.text:00541346                 pop     ebp
.text:00541347                 retn
.text:00541348 ; ---------------------------------------------------------------------------
.text:00541348
.text:00541348 loc_541348:                             ; CODE XREF: crypt_file+17
.text:00541348                 push    ebx
.text:00541349                 push    esi
.text:0054134A                 push    edi
.text:0054134B                 push    2               ; Origin
.text:0054134D                 push    0               ; Offset
.text:0054134F                 push    ebp             ; File
.text:00541350                 call    _fseek
.text:00541355                 push    ebp             ; File
```

```
.text:00541356                call    _ftell          ; get file size
.text:0054135B                push    0               ; Origin
.text:0054135D                push    0               ; Offset
.text:0054135F                push    ebp             ; File
.text:00541360                mov     [esp+2Ch+Str], eax
.text:00541364                call    _fseek          ; rewind to start
.text:00541369                mov     esi, [esp+2Ch+Str]
.text:0054136D                and     esi, 0FFFFFFC0h ; reset all lowest 6 bits
.text:00541370                add     esi, 40h        ; align size to 64-byte border
.text:00541373                push    esi             ; Size
.text:00541374                call    _malloc
.text:00541379                mov     ecx, esi
.text:0054137B                mov     ebx, eax        ; allocated buffer pointer -> to EBX
.text:0054137D                mov     edx, ecx
.text:0054137F                xor     eax, eax
.text:00541381                mov     edi, ebx
.text:00541383                push    ebp             ; File
.text:00541384                shr     ecx, 2
.text:00541387                rep stosd
.text:00541389                mov     ecx, edx
.text:0054138B                push    1               ; Count
.text:0054138D                and     ecx, 3
.text:00541390                rep stosb               ; memset (buffer, 0, aligned_size)
.text:00541392                mov     eax, [esp+38h+Str]
.text:00541396                push    eax             ; ElementSize
.text:00541397                push    ebx             ; DstBuf
.text:00541398                call    _fread          ; read file
.text:0054139D                push    ebp             ; File
.text:0054139E                call    _fclose
.text:005413A3                mov     ecx, [esp+44h+password]
.text:005413A7                push    ecx             ; password
.text:005413A8                push    esi             ; aligned size
.text:005413A9                push    ebx             ; buffer
.text:005413AA                call    crypt           ; do crypt
.text:005413AF                mov     edx, [esp+50h+Filename]
.text:005413B3                add     esp, 40h
.text:005413B6                push    offset aWb      ; "wb"
.text:005413BB                push    edx             ; Filename
.text:005413BC                call    _fopen
.text:005413C1                mov     edi, eax
.text:005413C3                push    edi             ; File
.text:005413C4                push    1               ; Count
.text:005413C6                push    3               ; Size
.text:005413C8                push    offset aQr9     ; "QR9"
.text:005413CD                call    _fwrite         ; write file signature
.text:005413D2                push    edi             ; File
.text:005413D3                push    1               ; Count
.text:005413D5                lea     eax, [esp+30h+Str]
.text:005413D9                push    4               ; Size
.text:005413DB                push    eax             ; Str
.text:005413DC                call    _fwrite         ; write original file size
.text:005413E1                push    edi             ; File
.text:005413E2                push    1               ; Count
.text:005413E4                push    esi             ; Size
.text:005413E5                push    ebx             ; Str
.text:005413E6                call    _fwrite         ; write crypted file
.text:005413EB                push    edi             ; File
.text:005413EC                call    _fclose
.text:005413F1                push    ebx             ; Memory
.text:005413F2                call    _free
.text:005413F7                add     esp, 40h
.text:005413FA                pop     edi
.text:005413FB                pop     esi
.text:005413FC                pop     ebx
.text:005413FD                pop     ebp
.text:005413FE                retn
```

```
.text:005413FE crypt_file      endp
.text:005413FE
.text:005413FE ; ---------------------------------------------------------------------------
.text:005413FF                 align 10h
.text:00541400
.text:00541400 ; =============== S U B R O U T I N E =======================================
.text:00541400
.text:00541400
.text:00541400 ; int __cdecl decrypt_file(char *Filename, int, void *Src)
.text:00541400 decrypt_file    proc near               ; CODE XREF: _main+6E
.text:00541400
.text:00541400 Filename        = dword ptr  4
.text:00541400 arg_4           = dword ptr  8
.text:00541400 Src             = dword ptr  0Ch
.text:00541400
.text:00541400                 mov     eax, [esp+Filename]
.text:00541404                 push    ebx
.text:00541405                 push    ebp
.text:00541406                 push    esi
.text:00541407                 push    edi
.text:00541408                 push    offset aRb      ; "rb"
.text:0054140D                 push    eax             ; Filename
.text:0054140E                 call    _fopen
.text:00541413                 mov     esi, eax
.text:00541415                 add     esp, 8
.text:00541418                 test    esi, esi
.text:0054141A                 jnz     short loc_54142E
.text:0054141C                 push    offset aCannotOpenIn_0 ; "Cannot open input file!\n"
.text:00541421                 call    _printf
.text:00541426                 add     esp, 4
.text:00541429                 pop     edi
.text:0054142A                 pop     esi
.text:0054142B                 pop     ebp
.text:0054142C                 pop     ebx
.text:0054142D                 retn
.text:0054142E ; ---------------------------------------------------------------------------
.text:0054142E
.text:0054142E loc_54142E:                             ; CODE XREF: decrypt_file+1A
.text:0054142E                 push    2               ; Origin
.text:00541430                 push    0               ; Offset
.text:00541432                 push    esi             ; File
.text:00541433                 call    _fseek
.text:00541438                 push    esi             ; File
.text:00541439                 call    _ftell
.text:0054143E                 push    0               ; Origin
.text:00541440                 push    0               ; Offset
.text:00541442                 push    esi             ; File
.text:00541443                 mov     ebp, eax
.text:00541445                 call    _fseek
.text:0054144A                 push    ebp             ; Size
.text:0054144B                 call    _malloc
.text:00541450                 push    esi             ; File
.text:00541451                 mov     ebx, eax
.text:00541453                 push    1               ; Count
.text:00541455                 push    ebp             ; ElementSize
.text:00541456                 push    ebx             ; DstBuf
.text:00541457                 call    _fread
.text:0054145C                 push    esi             ; File
.text:0054145D                 call    _fclose
.text:00541462                 add     esp, 34h
.text:00541465                 mov     ecx, 3
.text:0054146A                 mov     edi, offset aQr9_0 ; "QR9"
.text:0054146F                 mov     esi, ebx
.text:00541471                 xor     edx, edx
.text:00541473                 repe cmpsb
.text:00541475                 jz      short loc_541489
```

```
.text:00541477                push    offset aFileIsNotCrypt ; "File is not crypted!\n"
.text:0054147C                call    _printf
.text:00541481                add     esp, 4
.text:00541484                pop     edi
.text:00541485                pop     esi
.text:00541486                pop     ebp
.text:00541487                pop     ebx
.text:00541488                retn
.text:00541489 ; ---------------------------------------------------------------------------
.text:00541489
.text:00541489 loc_541489:                            ; CODE XREF: decrypt_file+75
.text:00541489                mov     eax, [esp+10h+Src]
.text:0054148D                mov     edi, [ebx+3]
.text:00541490                add     ebp, 0FFFFFFF9h
.text:00541493                lea     esi, [ebx+7]
.text:00541496                push    eax             ; Src
.text:00541497                push    ebp             ; int
.text:00541498                push    esi             ; int
.text:00541499                call    decrypt
.text:0054149E                mov     ecx, [esp+1Ch+arg_4]
.text:005414A2                push    offset aWb_0    ; "wb"
.text:005414A7                push    ecx             ; Filename
.text:005414A8                call    _fopen
.text:005414AD                mov     ebp, eax
.text:005414AF                push    ebp             ; File
.text:005414B0                push    1               ; Count
.text:005414B2                push    edi             ; Size
.text:005414B3                push    esi             ; Str
.text:005414B4                call    _fwrite
.text:005414B9                push    ebp             ; File
.text:005414BA                call    _fclose
.text:005414BF                push    ebx             ; Memory
.text:005414C0                call    _free
.text:005414C5                add     esp, 2Ch
.text:005414C8                pop     edi
.text:005414C9                pop     esi
.text:005414CA                pop     ebp
.text:005414CB                pop     ebx
.text:005414CC                retn
.text:005414CC decrypt_file   endp
```

All function and label names are given by me while analysis.
I started from top. Here is a function taking two file names and password.

```
.text:00541320 ; int __cdecl crypt_file(int Str, char *Filename, int password)
.text:00541320 crypt_file     proc near
.text:00541320
.text:00541320 Str            = dword ptr  4
.text:00541320 Filename       = dword ptr  8
.text:00541320 password       = dword ptr  0Ch
.text:00541320
```

Open file and report error in case of error:

```
.text:00541320                mov     eax, [esp+Str]
.text:00541324                push    ebp
.text:00541325                push    offset Mode     ; "rb"
.text:0054132A                push    eax             ; Filename
.text:0054132B                call    _fopen          ; open file
.text:00541330                mov     ebp, eax
.text:00541332                add     esp, 8
.text:00541335                test    ebp, ebp
.text:00541337                jnz     short loc_541348
.text:00541339                push    offset Format   ; "Cannot open input file!\n"
.text:0054133E                call    _printf
.text:00541343                add     esp, 4
.text:00541346                pop     ebp
```

```
.text:00541347                 retn
.text:00541348 ; ---------------------------------------------------------------------------
.text:00541348
.text:00541348 loc_541348:
```

Get file size via `fseek()`/`ftell()`:

```
.text:00541348 push    ebx
.text:00541349 push    esi
.text:0054134A push    edi
.text:0054134B push    2               ; Origin
.text:0054134D push    0               ; Offset
.text:0054134F push    ebp             ; File

; move current file position to the end
.text:00541350 call    _fseek
.text:00541355 push    ebp             ; File
.text:00541356 call    _ftell          ; get current file position
.text:0054135B push    0               ; Origin
.text:0054135D push    0               ; Offset
.text:0054135F push    ebp             ; File
.text:00541360 mov     [esp+2Ch+Str], eax

; move current file position to the start
.text:00541364 call    _fseek
```

This fragment of code calculates file size aligned to 64-byte border. This is because this cryptoalgorithm works with only 64-byte blocks. Its operation is pretty simple: divide file size by 64, forget about remainder and add 1, then multiple by 64. The following code removes remainder as if value was already divided by 64 and adds 64. It is almost the same.

```
.text:00541369 mov     esi, [esp+2Ch+Str]
.text:0054136D and     esi, 0FFFFFFC0h ; reset all lowest 6 bits
.text:00541370 add     esi, 40h        ; align size to 64-byte border
```

Allocate buffer with aligned size:

```
.text:00541373                 push    esi             ; Size
.text:00541374                 call    _malloc
```

Call memset(), e,g, clear allocated buffer[2].

```
.text:00541379 mov     ecx, esi
.text:0054137B mov     ebx, eax        ; allocated buffer pointer -> to EBX
.text:0054137D mov     edx, ecx
.text:0054137F xor     eax, eax
.text:00541381 mov     edi, ebx
.text:00541383 push    ebp             ; File
.text:00541384 shr     ecx, 2
.text:00541387 rep stosd
.text:00541389 mov     ecx, edx
.text:0054138B push    1               ; Count
.text:0054138D and     ecx, 3
.text:00541390 rep stosb               ; memset (buffer, 0, aligned_size)
```

Read file via standard C function `fread()`.

```
.text:00541392                 mov     eax, [esp+38h+Str]
.text:00541396                 push    eax             ; ElementSize
.text:00541397                 push    ebx             ; DstBuf
.text:00541398                 call    _fread          ; read file
.text:0054139D                 push    ebp             ; File
.text:0054139E                 call    _fclose
```

Call `crypt()`. This function takes buffer, buffer size (aligned) and password string.

[2] malloc() + memset() could be replaced by calloc()

```
.text:005413A3                mov     ecx, [esp+44h+password]
.text:005413A7                push    ecx             ; password
.text:005413A8                push    esi             ; aligned size
.text:005413A9                push    ebx             ; buffer
.text:005413AA                call    crypt           ; do crypt
```

Create output file. By the way, developer forgot to check if it is was created correctly! File opening result is being checked though.

```
.text:005413AF                mov     edx, [esp+50h+Filename]
.text:005413B3                add     esp, 40h
.text:005413B6                push    offset aWb      ; "wb"
.text:005413BB                push    edx             ; Filename
.text:005413BC                call    _fopen
.text:005413C1                mov     edi, eax
```

Newly created file handle is in EDI register now. Write signature “QR9”.

```
.text:005413C3                push    edi             ; File
.text:005413C4                push    1               ; Count
.text:005413C6                push    3               ; Size
.text:005413C8                push    offset aQr9     ; "QR9"
.text:005413CD                call    _fwrite         ; write file signature
```

Write actual file size (not aligned):

```
.text:005413D2                push    edi             ; File
.text:005413D3                push    1               ; Count
.text:005413D5                lea     eax, [esp+30h+Str]
.text:005413D9                push    4               ; Size
.text:005413DB                push    eax             ; Str
.text:005413DC                call    _fwrite         ; write original file size
```

Write crypted buffer:

```
.text:005413E1                push    edi             ; File
.text:005413E2                push    1               ; Count
.text:005413E4                push    esi             ; Size
.text:005413E5                push    ebx             ; Str
.text:005413E6                call    _fwrite         ; write crypted file
```

Close file and free allocated buffer:

```
.text:005413EB                push    edi             ; File
.text:005413EC                call    _fclose
.text:005413F1                push    ebx             ; Memory
.text:005413F2                call    _free
.text:005413F7                add     esp, 40h
.text:005413FA                pop     edi
.text:005413FB                pop     esi
.text:005413FC                pop     ebx
.text:005413FD                pop     ebp
.text:005413FE                retn
.text:005413FE crypt_file     endp
```

Here is reconstructed C-code:

```
void crypt_file(char *fin, char* fout, char *pw)
{
        FILE *f;
        int flen, flen_aligned;
        BYTE *buf;

        f=fopen(fin, "rb");

        if (f==NULL)
        {
```

```
                printf ("Cannot open input file!\n");
                return;
        };

        fseek (f, 0, SEEK_END);
        flen=ftell (f);
        fseek (f, 0, SEEK_SET);

        flen_aligned=(flen&0xFFFFFFC0)+0x40;

        buf=(BYTE*)malloc (flen_aligned);
        memset (buf, 0, flen_aligned);

        fread (buf, flen, 1, f);

        fclose (f);

        crypt (buf, flen_aligned, pw);

        f=fopen(fout, "wb");

        fwrite ("QR9", 3, 1, f);
        fwrite (&flen, 4, 1, f);
        fwrite (buf, flen_aligned, 1, f);

        fclose (f);

        free (buf);
};
```

Decrypting procedure is almost the same:

```
.text:00541400 ; int __cdecl decrypt_file(char *Filename, int, void *Src)
.text:00541400 decrypt_file    proc near
.text:00541400
.text:00541400 Filename        = dword ptr  4
.text:00541400 arg_4           = dword ptr  8
.text:00541400 Src             = dword ptr  0Ch
.text:00541400
.text:00541400                 mov     eax, [esp+Filename]
.text:00541404                 push    ebx
.text:00541405                 push    ebp
.text:00541406                 push    esi
.text:00541407                 push    edi
.text:00541408                 push    offset aRb      ; "rb"
.text:0054140D                 push    eax             ; Filename
.text:0054140E                 call    _fopen
.text:00541413                 mov     esi, eax
.text:00541415                 add     esp, 8
.text:00541418                 test    esi, esi
.text:0054141A                 jnz     short loc_54142E
.text:0054141C                 push    offset aCannotOpenIn_0 ; "Cannot open input file!\n"
.text:00541421                 call    _printf
.text:00541426                 add     esp, 4
.text:00541429                 pop     edi
.text:0054142A                 pop     esi
.text:0054142B                 pop     ebp
.text:0054142C                 pop     ebx
.text:0054142D                 retn
.text:0054142E ; ---------------------------------------------------------------------------
.text:0054142E
.text:0054142E loc_54142E:
.text:0054142E                 push    2               ; Origin
.text:00541430                 push    0               ; Offset
.text:00541432                 push    esi             ; File
.text:00541433                 call    _fseek
.text:00541438                 push    esi             ; File
```

```
.text:00541439                call    _ftell
.text:0054143E                push    0               ; Origin
.text:00541440                push    0               ; Offset
.text:00541442                push    esi             ; File
.text:00541443                mov     ebp, eax
.text:00541445                call    _fseek
.text:0054144A                push    ebp             ; Size
.text:0054144B                call    _malloc
.text:00541450                push    esi             ; File
.text:00541451                mov     ebx, eax
.text:00541453                push    1               ; Count
.text:00541455                push    ebp             ; ElementSize
.text:00541456                push    ebx             ; DstBuf
.text:00541457                call    _fread
.text:0054145C                push    esi             ; File
.text:0054145D                call    _fclose
```

Check signature (first 3 bytes):

```
.text:00541462                add     esp, 34h
.text:00541465                mov     ecx, 3
.text:0054146A                mov     edi, offset aQr9_0 ; "QR9"
.text:0054146F                mov     esi, ebx
.text:00541471                xor     edx, edx
.text:00541473                repe cmpsb
.text:00541475                jz      short loc_541489
```

Report an error if signature is absent:

```
.text:00541477                push    offset aFileIsNotCrypt ; "File is not crypted!\n"
.text:0054147C                call    _printf
.text:00541481                add     esp, 4
.text:00541484                pop     edi
.text:00541485                pop     esi
.text:00541486                pop     ebp
.text:00541487                pop     ebx
.text:00541488                retn
.text:00541489 ; ---------------------------------------------------------------------------
.text:00541489
.text:00541489 loc_541489:
```

Call `decrypt()`.

```
.text:00541489                mov     eax, [esp+10h+Src]
.text:0054148D                mov     edi, [ebx+3]
.text:00541490                add     ebp, 0FFFFFFF9h
.text:00541493                lea     esi, [ebx+7]
.text:00541496                push    eax             ; Src
.text:00541497                push    ebp             ; int
.text:00541498                push    esi             ; int
.text:00541499                call    decrypt
.text:0054149E                mov     ecx, [esp+1Ch+arg_4]
.text:005414A2                push    offset aWb_0    ; "wb"
.text:005414A7                push    ecx             ; Filename
.text:005414A8                call    _fopen
.text:005414AD                mov     ebp, eax
.text:005414AF                push    ebp             ; File
.text:005414B0                push    1               ; Count
.text:005414B2                push    edi             ; Size
.text:005414B3                push    esi             ; Str
.text:005414B4                call    _fwrite
.text:005414B9                push    ebp             ; File
.text:005414BA                call    _fclose
.text:005414BF                push    ebx             ; Memory
.text:005414C0                call    _free
.text:005414C5                add     esp, 2Ch
.text:005414C8                pop     edi
```

```
.text:005414C9                 pop     esi
.text:005414CA                 pop     ebp
.text:005414CB                 pop     ebx
.text:005414CC                 retn
.text:005414CC decrypt_file    endp
```

Here is reconstructed C-code:

```
void decrypt_file(char *fin, char* fout, char *pw)
{
        FILE *f;
        int real_flen, flen;
        BYTE *buf;

        f=fopen(fin, "rb");

        if (f==NULL)
        {
                printf ("Cannot open input file!\n");
                return;
        };

        fseek (f, 0, SEEK_END);
        flen=ftell (f);
        fseek (f, 0, SEEK_SET);

        buf=(BYTE*)malloc (flen);

        fread (buf, flen, 1, f);

        fclose (f);

        if (memcmp (buf, "QR9", 3)!=0)
        {
                printf ("File is not crypted!\n");
                return;
        };

        memcpy (&real_flen, buf+3, 4);

        decrypt (buf+(3+4), flen-(3+4), pw);

        f=fopen(fout, "wb");

        fwrite (buf+(3+4), real_flen, 1, f);

        fclose (f);

        free (buf);
};
```

OK, now let's go deeper.

Function `crypt()`:

```
.text:00541260 crypt           proc near
.text:00541260
.text:00541260 arg_0           = dword ptr  4
.text:00541260 arg_4           = dword ptr  8
.text:00541260 arg_8           = dword ptr  0Ch
.text:00541260
.text:00541260                 push    ebx
.text:00541261                 mov     ebx, [esp+4+arg_0]
.text:00541265                 push    ebp
.text:00541266                 push    esi
.text:00541267                 push    edi
.text:00541268                 xor     ebp, ebp
.text:0054126A
```

```
.text:0054126A loc_54126A:
```

This fragment of code copies part of input buffer to internal array I named later "cube64". The size is in ECX register. MOVSD means *move 32-bit dword*, so, 16 of 32-bit dwords are exactly 64 bytes.

```
.text:0054126A                mov     eax, [esp+10h+arg_8]
.text:0054126E                mov     ecx, 10h
.text:00541273                mov     esi, ebx   ; EBX is pointer within input buffer
.text:00541275                mov     edi, offset cube64
.text:0054127A                push    1
.text:0054127C                push    eax
.text:0054127D                rep movsd
```

Call rotate_all_with_password():

```
.text:0054127F                call    rotate_all_with_password
```

Copy crypted contents back from "cube64" to buffer:

```
.text:00541284                mov     eax, [esp+18h+arg_4]
.text:00541288                mov     edi, ebx
.text:0054128A                add     ebp, 40h
.text:0054128D                add     esp, 8
.text:00541290                mov     ecx, 10h
.text:00541295                mov     esi, offset cube64
.text:0054129A                add     ebx, 40h  ; add 64 to input buffer pointer
.text:0054129D                cmp     ebp, eax  ; EBP contain ammount of crypted data.
.text:0054129F                rep movsd
```

If EBP is not bigger that input argument size, then continue to next block.

```
.text:005412A1                jl      short loc_54126A
.text:005412A3                pop     edi
.text:005412A4                pop     esi
.text:005412A5                pop     ebp
.text:005412A6                pop     ebx
.text:005412A7                retn
.text:005412A7 crypt          endp
```

Reconstructed crypt() function:

```
void crypt (BYTE *buf, int sz, char *pw)
{
        int i=0;

        do
        {
                memcpy (cube, buf+i, 8*8);
                rotate_all (pw, 1);
                memcpy (buf+i, cube, 8*8);
                i+=64;
        }
        while (i<sz);
};
```

OK, now let's go deeper into function rotate_all_with_password(). It takes two arguments: password string and number. In crypt(), number 1 is used, and in decrypt() (where rotate_all_with_password() function is called too), number is 3.

```
.text:005411B0 rotate_all_with_password proc near
.text:005411B0
.text:005411B0 arg_0          = dword ptr  4
.text:005411B0 arg_4          = dword ptr  8
.text:005411B0
.text:005411B0                mov     eax, [esp+arg_0]
.text:005411B4                push    ebp
.text:005411B5                mov     ebp, eax
```

Check for character in password. If it is zero, exit:

```
.text:005411B7                cmp     byte ptr [eax], 0
.text:005411BA                jz      exit
.text:005411C0                push    ebx
.text:005411C1                mov     ebx, [esp+8+arg_4]
.text:005411C5                push    esi
.text:005411C6                push    edi
.text:005411C7
.text:005411C7 loop_begin:
```

Call `tolower()`, standard C function.

```
.text:005411C7                movsx   eax, byte ptr [ebp+0]
.text:005411CB                push    eax             ; C
.text:005411CC                call    _tolower
.text:005411D1                add     esp, 4
```

Hmm, if password contains non-alphabetical latin character, it is skipped! Indeed, if we run crypting utility and try non-alphabetical latin characters in password, they seem to be ignored.

```
.text:005411D4                cmp     al, 'a'
.text:005411D6                jl      short next_character_in_password
.text:005411D8                cmp     al, 'z'
.text:005411DA                jg      short next_character_in_password
.text:005411DC                movsx   ecx, al
```

Subtract "a" value (97) from character.

```
.text:005411DF                sub     ecx, 'a'  ; 97
```

After subtracting, we'll get 0 for "a" here, 1 for "b", etc. And 25 for "z".

```
.text:005411E2                cmp     ecx, 24
.text:005411E5                jle     short skip_subtracting
.text:005411E7                sub     ecx, 24
```

It seems, "y" and "z" are exceptional characters too. After that fragment of code, "y" becomes 0 and "z" — 1. This means, 26 latin alphabet symbols will become values in range 0..23, (24 in total).

```
.text:005411EA
.text:005411EA skip_subtracting:                      ; CODE XREF: rotate_all_with_password+35
```

This is actually division via multiplication. Read more about it in "Division by 9" section 1.12.
The code actually divides password character value by 3.

```
.text:005411EA                mov     eax, 55555556h
.text:005411EF                imul    ecx
.text:005411F1                mov     eax, edx
.text:005411F3                shr     eax, 1Fh
.text:005411F6                add     edx, eax
.text:005411F8                mov     eax, ecx
.text:005411FA                mov     esi, edx
.text:005411FC                mov     ecx, 3
.text:00541201                cdq
.text:00541202                idiv    ecx
```

EDX is the remainder of division.

```
.text:00541204 sub     edx, 0
.text:00541207 jz      short call_rotate1 ; if remainder is zero, go to rotate1
.text:00541209 dec     edx
.text:0054120A jz      short call_rotate2 ; .. it it is 1, go to rotate2
.text:0054120C dec     edx
.text:0054120D jnz     short next_character_in_password
.text:0054120F test    ebx, ebx
.text:00541211 jle     short next_character_in_password
.text:00541213 mov     edi, ebx
```

If remainder is 2, call `rotate3()`. `EDI` is a second argument of `rotate_all_with_password()`. As I already wrote, 1 is for crypting operations and 3 is for decrypting. So, here is a loop. When crypting, rotate1/2/3 will be called the same number of times as given in the first argument.

```
.text:00541215 call_rotate3:
.text:00541215                 push    esi
.text:00541216                 call    rotate3
.text:0054121B                 add     esp, 4
.text:0054121E                 dec     edi
.text:0054121F                 jnz     short call_rotate3
.text:00541221                 jmp     short next_character_in_password
.text:00541223
.text:00541223 call_rotate2:
.text:00541223                 test    ebx, ebx
.text:00541225                 jle     short next_character_in_password
.text:00541227                 mov     edi, ebx
.text:00541229
.text:00541229 loc_541229:
.text:00541229                 push    esi
.text:0054122A                 call    rotate2
.text:0054122F                 add     esp, 4
.text:00541232                 dec     edi
.text:00541233                 jnz     short loc_541229
.text:00541235                 jmp     short next_character_in_password
.text:00541237
.text:00541237 call_rotate1:
.text:00541237                 test    ebx, ebx
.text:00541239                 jle     short next_character_in_password
.text:0054123B                 mov     edi, ebx
.text:0054123D
.text:0054123D loc_54123D:
.text:0054123D                 push    esi
.text:0054123E                 call    rotate1
.text:00541243                 add     esp, 4
.text:00541246                 dec     edi
.text:00541247                 jnz     short loc_54123D
.text:00541249
```

Fetch next character from password string.

```
.text:00541249 next_character_in_password:
.text:00541249                 mov     al, [ebp+1]
```

Increment character pointer within password string:

```
.text:0054124C                 inc     ebp
.text:0054124D                 test    al, al
.text:0054124F                 jnz     loop_begin
.text:00541255                 pop     edi
.text:00541256                 pop     esi
.text:00541257                 pop     ebx
.text:00541258
.text:00541258 exit:
.text:00541258                 pop     ebp
.text:00541259                 retn
.text:00541259 rotate_all_with_password endp
```

Here is reconstructed C code:

```
void rotate_all (char *pwd, int v)
{
        char *p=pwd;

        while (*p)
        {
                char c=*p;
                int q;
```

```
                c=tolower (c);

                if (c>='a' && c<='z')
                {
                        q=c-'a';
                        if (q>24)
                                q-=24;

                        int quotient=q/3;
                        int remainder=q % 3;

                        switch (remainder)
                        {
                        case 0: for (int i=0; i<v; i++) rotate1 (quotient); break;
                        case 1: for (int i=0; i<v; i++) rotate2 (quotient); break;
                        case 2: for (int i=0; i<v; i++) rotate3 (quotient); break;
                        };
                };

                p++;
        };
};
```

Now let's go deeper and investigate rotate1/2/3 functions. Each function calls two another functions. I eventually gave them names `set_bit()` and `get_bit()`.

Let's start with `get_bit()`:

```
.text:00541050 get_bit         proc near
.text:00541050
.text:00541050 arg_0           = dword ptr  4
.text:00541050 arg_4           = dword ptr  8
.text:00541050 arg_8           = byte ptr  0Ch
.text:00541050
.text:00541050                 mov     eax, [esp+arg_4]
.text:00541054                 mov     ecx, [esp+arg_0]
.text:00541058                 mov     al, cube64[eax+ecx*8]
.text:0054105F                 mov     cl, [esp+arg_8]
.text:00541063                 shr     al, cl
.text:00541065                 and     al, 1
.text:00541067                 retn
.text:00541067 get_bit         endp
```

…in other words: calculate an index in the array cube64: *arg_4* + *arg_0* * *8*. Then shift a byte from an array by arg_8 bits right. Isolate lowest bit and return it.

Let's see another function, `set_bit()`:

```
.text:00541000 set_bit         proc near
.text:00541000
.text:00541000 arg_0           = dword ptr  4
.text:00541000 arg_4           = dword ptr  8
.text:00541000 arg_8           = dword ptr  0Ch
.text:00541000 arg_C           = byte ptr  10h
.text:00541000
.text:00541000                 mov     al, [esp+arg_C]
.text:00541004                 mov     ecx, [esp+arg_8]
.text:00541008                 push    esi
.text:00541009                 mov     esi, [esp+4+arg_0]
.text:0054100D                 test    al, al
.text:0054100F                 mov     eax, [esp+4+arg_4]
.text:00541013                 mov     dl, 1
.text:00541015                 jz      short loc_54102B
```

DL is 1 here. Shift left it by arg_8. For example, if arg_8 is 4, DL register value became 0x10 or 1000 in binary form.

```
.text:00541017                 shl     dl, cl
.text:00541019                 mov     cl, cube64[eax+esi*8]
```

Get bit from array and explicitly set one.

```
.text:00541020                 or      cl, dl
```

Store it back:

```
.text:00541022                 mov     cube64[eax+esi*8], cl
.text:00541029                 pop     esi
.text:0054102A                 retn
.text:0054102B ; ---------------------------------------------------------------------------
.text:0054102B
.text:0054102B loc_54102B:
.text:0054102B                 shl     dl, cl
```

If arg_C is not zero…

```
.text:0054102D                 mov     cl, cube64[eax+esi*8]
```

…invert DL. For example, if DL state after shift was 0x10 or 1000 in binary form, there will be 0xEF after `NOT` instruction or 11101111 in binary form.

```
.text:00541034                 not     dl
```

This instruction clears bit, in other words, it saves all bits in CL which are also set in DL except those in DL which are cleared. This means that if DL is, for example, 11101111 in binary form, all bits will be saved except 5th (counting from lowest bit).

```
.text:00541036                 and     cl, dl
```

Store it back:

```
.text:00541038                 mov     cube64[eax+esi*8], cl
.text:0054103F                 pop     esi
.text:00541040                 retn
.text:00541040 set_bit         endp
```

It is almost the same as `get_bit()`, except, if arg_C is zero, the function clears specific bit in array, or sets it otherwise.

We also know that array size is 64. First two arguments both in `set_bit()` and `get_bit()` could be seen as 2D cooridnates. Then array will be 8*8 matrix.

Here is C representation of what we already know:

```
#define IS_SET(flag, bit)       ((flag) & (bit))
#define SET_BIT(var, bit)       ((var) |= (bit))
#define REMOVE_BIT(var, bit)    ((var) &= ~(bit))

char cube[8][8];

void set_bit (int x, int y, int shift, int bit)
{
        if (bit)
                SET_BIT (cube[x][y], 1<<shift);
        else
                REMOVE_BIT (cube[x][y], 1<<shift);
};

int get_bit (int x, int y, int shift)
{
        if ((cube[x][y]>>shift)&1==1)
                return 1;
        return 0;
};
```

Now let's get back to rotate1/2/3 functions.

```
.text:00541070 rotate1         proc near
.text:00541070
```

Internal array allocation in local stack, its size 64 bytes:

```
.text:00541070 internal_array_64= byte ptr -40h
.text:00541070 arg_0           = dword ptr  4
.text:00541070
.text:00541070                 sub     esp, 40h
.text:00541073                 push    ebx
.text:00541074                 push    ebp
.text:00541075                 mov     ebp, [esp+48h+arg_0]
.text:00541079                 push    esi
.text:0054107A                 push    edi
.text:0054107B                 xor     edi, edi        ; EDI is loop1 counter
```

EBX is a pointer to internal array:

```
.text:0054107D                 lea     ebx, [esp+50h+internal_array_64]
.text:00541081
```

Two nested loops are here:

```
.text:00541081 first_loop1_begin:
.text:00541081    xor     esi, esi        ; ESI is loop 2 counter
.text:00541083
.text:00541083 first_loop2_begin:
.text:00541083    push    ebp             ; arg_0
.text:00541084    push    esi             ; loop 1 counter
.text:00541085    push    edi             ; loop 2 counter
.text:00541086    call    get_bit
.text:0054108B    add     esp, 0Ch
.text:0054108E    mov     [ebx+esi], al   ; store to internal array
.text:00541091    inc     esi             ; increment loop 1 counter
.text:00541092    cmp     esi, 8
.text:00541095    jl      short first_loop2_begin
.text:00541097    inc     edi             ; increment loop 2 counter
.text:00541098    add     ebx, 8          ; increment internal array pointer by 8 at each loop 1 iteration
.text:0054109B    cmp     edi, 8
.text:0054109E    jl      short first_loop1_begin
```

...we see that both loop counters are in range 0..7. Also, they are used as first and second arguments of `get_bit()`. Third argument of `get_bit()` is the only argument of `rotate1()`. What `get_bit()` returns, is being placed into internal array.

Prepare pointer to internal array again:

```
.text:005410A0    lea     ebx, [esp+50h+internal_array_64]
.text:005410A4    mov     edi, 7          ; EDI is loop 1 counter, initial state is 7
.text:005410A9
.text:005410A9 second_loop1_begin:
.text:005410A9    xor     esi, esi        ; ESI is loop 2 counter
.text:005410AB
.text:005410AB second_loop2_begin:
.text:005410AB    mov     al, [ebx+esi]   ; value from internal array
.text:005410AE    push    eax
.text:005410AF    push    ebp             ; arg_0
.text:005410B0    push    edi             ; loop 1 counter
.text:005410B1    push    esi             ; loop 2 counter
.text:005410B2    call    set_bit
.text:005410B7    add     esp, 10h
.text:005410BA    inc     esi             ; increment loop 2 counter
.text:005410BB    cmp     esi, 8
.text:005410BE    jl      short second_loop2_begin
.text:005410C0    dec     edi             ; decrement loop 2 counter
.text:005410C1    add     ebx, 8          ; increment pointer in internal array
```

```
.text:005410C4    cmp     edi, 0FFFFFFFFh
.text:005410C7    jg      short second_loop1_begin
.text:005410C9    pop     edi
.text:005410CA    pop     esi
.text:005410CB    pop     ebp
.text:005410CC    pop     ebx
.text:005410CD    add     esp, 40h
.text:005410D0    retn
.text:005410D0 rotate1          endp
```

...this code is placing contents from internal array to cube global array via `set_bit()` function, *but*, in different order! Now loop 1 counter is in range 7 to 0, decrementing at each iteration!

C code representation looks like:

```
void rotate1 (int v)
{
        bool tmp[8][8]; // internal array
        int i, j;

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        tmp[i][j]=get_bit (i, j, v);

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        set_bit (j, 7-i, v, tmp[x][y]);
};
```

Not very understandable, but if we will take a look at `rotate2()` function:

```
.text:005410E0 rotate2 proc near
.text:005410E0
.text:005410E0 internal_array_64 = byte ptr -40h
.text:005410E0 arg_0 = dword ptr  4
.text:005410E0
.text:005410E0     sub     esp, 40h
.text:005410E3     push    ebx
.text:005410E4     push    ebp
.text:005410E5     mov     ebp, [esp+48h+arg_0]
.text:005410E9     push    esi
.text:005410EA     push    edi
.text:005410EB     xor     edi, edi        ; loop 1 counter
.text:005410ED     lea     ebx, [esp+50h+internal_array_64]
.text:005410F1
.text:005410F1 loc_5410F1:
.text:005410F1     xor     esi, esi        ; loop 2 counter
.text:005410F3
.text:005410F3 loc_5410F3:
.text:005410F3     push    esi             ; loop 2 counter
.text:005410F4     push    edi             ; loop 1 counter
.text:005410F5     push    ebp             ; arg_0
.text:005410F6     call    get_bit
.text:005410FB     add     esp, 0Ch
.text:005410FE     mov     [ebx+esi], al   ; store to internal array
.text:00541101     inc     esi             ; increment loop 1 counter
.text:00541102     cmp     esi, 8
.text:00541105     jl      short loc_5410F3
.text:00541107     inc     edi             ; increment loop 2 counter
.text:00541108     add     ebx, 8
.text:0054110B     cmp     edi, 8
.text:0054110E     jl      short loc_5410F1
.text:00541110     lea     ebx, [esp+50h+internal_array_64]
.text:00541114     mov     edi, 7          ; loop 1 counter is initial state 7
.text:00541119
.text:00541119 loc_541119:
.text:00541119     xor     esi, esi        ; loop 2 counter
.text:0054111B
```

```
.text:0054111B loc_54111B:
.text:0054111B     mov     al, [ebx+esi]   ; get byte from internal array
.text:0054111E     push    eax
.text:0054111F     push    edi             ; loop 1 counter
.text:00541120     push    esi             ; loop 2 counter
.text:00541121     push    ebp             ; arg_0
.text:00541122     call    set_bit
.text:00541127     add     esp, 10h
.text:0054112A     inc     esi             ; increment loop 2 counter
.text:0054112B     cmp     esi, 8
.text:0054112E     jl      short loc_54111B
.text:00541130     dec     edi             ; decrement loop 2 counter
.text:00541131     add     ebx, 8
.text:00541134     cmp     edi, 0FFFFFFFFh
.text:00541137     jg      short loc_541119
.text:00541139     pop     edi
.text:0054113A     pop     esi
.text:0054113B     pop     ebp
.text:0054113C     pop     ebx
.text:0054113D     add     esp, 40h
.text:00541140     retn
.text:00541140 rotate2 endp
```

It is *almost* the same, except of different order of arguments of `get_bit()` and `set_bit()`. Let's rewrite it in C-like code:

```
void rotate2 (int v)
{
        bool tmp[8][8]; // internal array
        int i, j;

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        tmp[i][j]=get_bit (v, i, j);

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        set_bit (v, j, 7-i, tmp[i][j]);
};
```

Let's also rewrite `rotate3()` function:

```
void rotate3 (int v)
{
        bool tmp[8][8];
        int i, j;

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        tmp[i][j]=get_bit (i, v, j);

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        set_bit (7-j, v, i, tmp[i][j]);
};
```

Well, now things are simpler. If we consider cube64 as 3D cube 8*8*8, where each element is bit, `get_bit()` and `set_bit()` take just coordinates of bit on input.

rotate1/2/3 functions are in fact rotating all bits in specific plane. Three functions are each for each cube side and v argument is setting plane in range 0..7.

Maybe, algorithm's author was thinking of 8*8*8 Rubik's cube?!

Yes, indeed.

Let's get closer into `decrypt()` function, I rewrote it here:

```
void decrypt (BYTE *buf, int sz, char *pw)
{
```

```
        char *p=strdup (pw);
        strrev (p);
        int i=0;

        do
        {
                memcpy (cube, buf+i, 8*8);
                rotate_all (p, 3);
                memcpy (buf+i, cube, 8*8);
                i+=64;
        }
        while (i<sz);

        free (p);
};
```

It is almost the same excert of `crypt()`, *but* password string is reversed by strrev() standard C function and `rotate_all()` is called with argument 3.

This means, that in case of decryption, each corresponding rotate1/2/3 call will be performed thrice.

This is almost as in Rubik'c cube! If you want to get back, do the same in reverse order and direction! If you need to undo effect of rotating one place in clockwise direction, rotate it thrice in counter-clockwise direction.

`rotate1()` is apparently for rotating "front" plane. `rotate2()` is apparently for rotating "top" plane. `rotate3()` is apparently for rotating "left" plane.

Let's get back to core of `rotate_all()` function:

```
q=c-'a';
if (q>24)
        q-=24;

int quotient=q/3; // in range 0..7
int remainder=q % 3;

switch (remainder)
{
    case 0: for (int i=0; i<v; i++) rotate1 (quotient); break; // front
    case 1: for (int i=0; i<v; i++) rotate2 (quotient); break; // top
    case 2: for (int i=0; i<v; i++) rotate3 (quotient); break; // left
};
```

Now it is much simpler to understand: each password character defines side (one of three) and plane (one of 8). 3*8 = 24, that's why two last characters of latin alphabet are remapped to fit an alphabet of exactly 24 elements.

The algorithm is clearly weak: in case of short passwords, one can see, that in crypted file there are some original bytes of original file in binary files editor.

Here is reconstructed whole source code:

```
#include <windows.h>

#include <stdio.h>
#include <assert.h>

#define IS_SET(flag, bit)       ((flag) & (bit))
#define SET_BIT(var, bit)       ((var) |= (bit))
#define REMOVE_BIT(var, bit)    ((var) &= ~(bit))

static BYTE cube[8][8];

void set_bit (int x, int y, int z, bool bit)
{
        if (bit)
                SET_BIT (cube[x][y], 1<<z);
        else
                REMOVE_BIT (cube[x][y], 1<<z);
};
```

```
bool get_bit (int x, int y, int z)
{
        if ((cube[x][y]>>z)&1==1)
                return true;
        return false;
};

void rotate_f (int row)
{
        bool tmp[8][8];
        int x, y;

        for (x=0; x<8; x++)
                for (y=0; y<8; y++)
                        tmp[x][y]=get_bit (x, y, row);

        for (x=0; x<8; x++)
                for (y=0; y<8; y++)
                        set_bit (y, 7-x, row, tmp[x][y]);
};

void rotate_t (int row)
{
        bool tmp[8][8];
        int y, z;

        for (y=0; y<8; y++)
                for (z=0; z<8; z++)
                        tmp[y][z]=get_bit (row, y, z);

        for (y=0; y<8; y++)
                for (z=0; z<8; z++)
                        set_bit (row, z, 7-y, tmp[y][z]);
};

void rotate_l (int row)
{
        bool tmp[8][8];
        int x, z;

        for (x=0; x<8; x++)
                for (z=0; z<8; z++)
                        tmp[x][z]=get_bit (x, row, z);

        for (x=0; x<8; x++)
                for (z=0; z<8; z++)
                        set_bit (7-z, row, x, tmp[x][z]);
};

void rotate_all (char *pwd, int v)
{
        char *p=pwd;

        while (*p)
        {
                char c=*p;
                int q;

                c=tolower (c);

                if (c>='a' && c<='z')
                {
                        q=c-'a';
                        if (q>24)
                                q-=24;
```

```
                        int quotient=q/3;
                        int remainder=q % 3;

                        switch (remainder)
                        {
                        case 0: for (int i=0; i<v; i++) rotate1 (quotient); break;
                        case 1: for (int i=0; i<v; i++) rotate2 (quotient); break;
                        case 2: for (int i=0; i<v; i++) rotate3 (quotient); break;
                        };
                };

                p++;
        };
};

void crypt (BYTE *buf, int sz, char *pw)
{
        int i=0;

        do
        {
                memcpy (cube, buf+i, 8*8);
                rotate_all (pw, 1);
                memcpy (buf+i, cube, 8*8);
                i+=64;
        }
        while (i<sz);
};

void decrypt (BYTE *buf, int sz, char *pw)
{
        char *p=strdup (pw);
        strrev (p);
        int i=0;

        do
        {
                memcpy (cube, buf+i, 8*8);
                rotate_all (p, 3);
                memcpy (buf+i, cube, 8*8);
                i+=64;
        }
        while (i<sz);

        free (p);
};

void crypt_file(char *fin, char* fout, char *pw)
{
        FILE *f;
        int flen, flen_aligned;
        BYTE *buf;

        f=fopen(fin, "rb");

        if (f==NULL)
        {
                printf ("Cannot open input file!\n");
                return;
        };

        fseek (f, 0, SEEK_END);
        flen=ftell (f);
        fseek (f, 0, SEEK_SET);

        flen_aligned=(flen&0xFFFFFFC0)+0x40;
```

```
        buf=(BYTE*)malloc (flen_aligned);
        memset (buf, 0, flen_aligned);

        fread (buf, flen, 1, f);

        fclose (f);

        crypt (buf, flen_aligned, pw);

        f=fopen(fout, "wb");

        fwrite ("QR9", 3, 1, f);
        fwrite (&flen, 4, 1, f);
        fwrite (buf, flen_aligned, 1, f);

        fclose (f);

        free (buf);

};

void decrypt_file(char *fin, char* fout, char *pw)
{
        FILE *f;
        int real_flen, flen;
        BYTE *buf;

        f=fopen(fin, "rb");

        if (f==NULL)
        {
                printf ("Cannot open input file!\n");
                return;
        };

        fseek (f, 0, SEEK_END);
        flen=ftell (f);
        fseek (f, 0, SEEK_SET);

        buf=(BYTE*)malloc (flen);

        fread (buf, flen, 1, f);

        fclose (f);

        if (memcmp (buf, "QR9", 3)!=0)
        {
                printf ("File is not crypted!\n");
                return;
        };

        memcpy (&real_flen, buf+3, 4);

        decrypt (buf+(3+4), flen-(3+4), pw);

        f=fopen(fout, "wb");

        fwrite (buf+(3+4), real_flen, 1, f);

        fclose (f);

        free (buf);
};

// run: input output 0/1 password
```

```
// 0 for encrypt, 1 for decrypt

int main(int argc, char *argv[])
{
        if (argc!=5)
        {
                printf ("Incorrect parameters!\n");
                return 1;
        };

        if (strcmp (argv[3], "0")==0)
                crypt_file (argv[1], argv[2], argv[4]);
        else
                if (strcmp (argv[3], "1")==0)
                        decrypt_file (argv[1], argv[2], argv[4]);
                else
                        printf ("Wrong param %s\n", argv[3]);

        return 0;
};
```

7.2 SAP

7.2.1 About SAP client network traffic compression

(Tracing connection between TDW_NOCOMPRESS SAPGUI[3] environment variable to nagging pop-up window and actual data compression routine.)

It's known that network traffic between SAPGUI and SAP is not crypted by default, it's rather compressed (read here and here).

It's also known that by setting environment variable *TDW_NOCOMPRESS* to 1, it's possible to turn network packets compression off.

But you will see a nagging pop-up windows that cannot be closed:

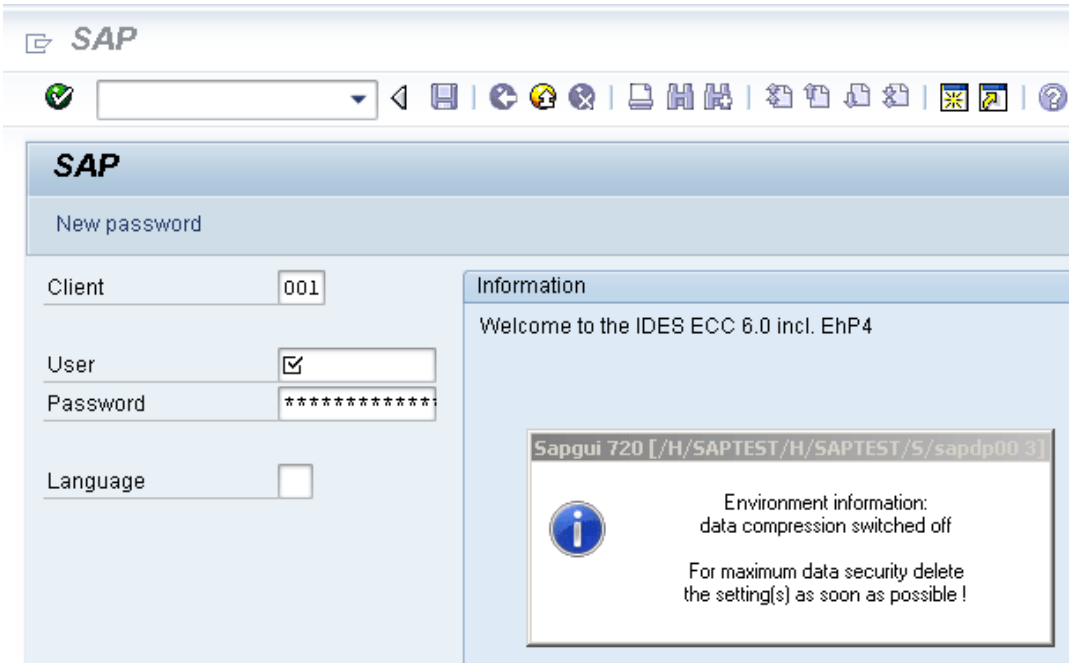

Figure 7.1: Screenshot

Let's see, if we can remove that window somehow.

[3] SAP GUI client

But before this, let's see what we already know. First: we know that environment variable *TDW_NOCOMPRESS* is checked somewhere inside of SAPGUI client. Second: string like "data compression switched off" must be present somewhere too. With the help of FAR file manager I found that both of these strings are stored in the SAPguilib.dll file.

So let's open SAPguilib.dll in IDA 5 and search for *"TDW_NOCOMPRESS"* string. Yes, it is present and there is only one reference to it.

We see the following fragment of code (all file offsets are valid for SAPGUI 720 win32, SAPguilib.dll file version 7200,1,0,9009):

```
.text:6440D51B                 lea     eax, [ebp+2108h+var_211C]
.text:6440D51E                 push    eax             ; int
.text:6440D51F                 push    offset aTdw_nocompress ; "TDW_NOCOMPRESS"
.text:6440D524                 mov     byte ptr [edi+15h], 0
.text:6440D528                 call    chk_env
.text:6440D52D                 pop     ecx
.text:6440D52E                 pop     ecx
.text:6440D52F                 push    offset byte_64443AF8
.text:6440D534                 lea     ecx, [ebp+2108h+var_211C]

; demangled name: int ATL::CStringT::Compare(char const *)const
.text:6440D537                 call    ds:mfc90_1603
.text:6440D53D                 test    eax, eax
.text:6440D53F                 jz      short loc_6440D55A
.text:6440D541                 lea     ecx, [ebp+2108h+var_211C]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:6440D544                 call    ds:mfc90_910
.text:6440D54A                 push    eax             ; Str
.text:6440D54B                 call    ds:atoi
.text:6440D551                 test    eax, eax
.text:6440D553                 setnz   al
.text:6440D556                 pop     ecx
.text:6440D557                 mov     [edi+15h], al
```

String returned by `chk_env()` via second argument is then handled by MFC string functions and then `atoi()`[4] is called. After that, numerical value is stored to `edi+15h`.

Also, take a look onto `chk_env()` function (I gave a name to it):

```
.text:64413F20 ; int __cdecl chk_env(char *VarName, int)
.text:64413F20 chk_env         proc near
.text:64413F20
.text:64413F20 DstSize         = dword ptr -0Ch
.text:64413F20 var_8           = dword ptr -8
.text:64413F20 DstBuf          = dword ptr -4
.text:64413F20 VarName         = dword ptr  8
.text:64413F20 arg_4           = dword ptr  0Ch
.text:64413F20
.text:64413F20                 push    ebp
.text:64413F21                 mov     ebp, esp
.text:64413F23                 sub     esp, 0Ch
.text:64413F26                 mov     [ebp+DstSize], 0
.text:64413F2D                 mov     [ebp+DstBuf], 0
.text:64413F34                 push    offset unk_6444C88C
.text:64413F39                 mov     ecx, [ebp+arg_4]

; (demangled name) ATL::CStringT::operator=(char const *)
.text:64413F3C                 call    ds:mfc90_820
.text:64413F42                 mov     eax, [ebp+VarName]
.text:64413F45                 push    eax             ; VarName
.text:64413F46                 mov     ecx, [ebp+DstSize]
.text:64413F49                 push    ecx             ; DstSize
.text:64413F4A                 mov     edx, [ebp+DstBuf]
.text:64413F4D                 push    edx             ; DstBuf
```

[4] standard C library function, coverting number in string into number

```
.text:64413F4E                 lea     eax, [ebp+DstSize]
.text:64413F51                 push    eax             ; ReturnSize
.text:64413F52                 call    ds:getenv_s
.text:64413F58                 add     esp, 10h
.text:64413F5B                 mov     [ebp+var_8], eax
.text:64413F5E                 cmp     [ebp+var_8], 0
.text:64413F62                 jz      short loc_64413F68
.text:64413F64                 xor     eax, eax
.text:64413F66                 jmp     short loc_64413FBC
.text:64413F68 ; ---------------------------------------------------------------------------
.text:64413F68
.text:64413F68 loc_64413F68:
.text:64413F68                 cmp     [ebp+DstSize], 0
.text:64413F6C                 jnz     short loc_64413F72
.text:64413F6E                 xor     eax, eax
.text:64413F70                 jmp     short loc_64413FBC
.text:64413F72 ; ---------------------------------------------------------------------------
.text:64413F72
.text:64413F72 loc_64413F72:
.text:64413F72                 mov     ecx, [ebp+DstSize]
.text:64413F75                 push    ecx
.text:64413F76                 mov     ecx, [ebp+arg_4]

; demangled name: ATL::CSimpleStringT<char, 1>::Preallocate(int)
.text:64413F79                 call    ds:mfc90_2691
.text:64413F7F                 mov     [ebp+DstBuf], eax
.text:64413F82                 mov     edx, [ebp+VarName]
.text:64413F85                 push    edx             ; VarName
.text:64413F86                 mov     eax, [ebp+DstSize]
.text:64413F89                 push    eax             ; DstSize
.text:64413F8A                 mov     ecx, [ebp+DstBuf]
.text:64413F8D                 push    ecx             ; DstBuf
.text:64413F8E                 lea     edx, [ebp+DstSize]
.text:64413F91                 push    edx             ; ReturnSize
.text:64413F92                 call    ds:getenv_s
.text:64413F98                 add     esp, 10h
.text:64413F9B                 mov     [ebp+var_8], eax
.text:64413F9E                 push    0FFFFFFFFh
.text:64413FA0                 mov     ecx, [ebp+arg_4]

; demangled name: ATL::CSimpleStringT::ReleaseBuffer(int)
.text:64413FA3                 call    ds:mfc90_5835
.text:64413FA9                 cmp     [ebp+var_8], 0
.text:64413FAD                 jz      short loc_64413FB3
.text:64413FAF                 xor     eax, eax
.text:64413FB1                 jmp     short loc_64413FBC
.text:64413FB3 ; ---------------------------------------------------------------------------
.text:64413FB3
.text:64413FB3 loc_64413FB3:
.text:64413FB3                 mov     ecx, [ebp+arg_4]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:64413FB6                 call    ds:mfc90_910
.text:64413FBC
.text:64413FBC loc_64413FBC:
.text:64413FBC
.text:64413FBC                 mov     esp, ebp
.text:64413FBE                 pop     ebp
.text:64413FBF                 retn
.text:64413FBF chk_env         endp
```

Yes. `getenv_s()`[5] function is Microsoft security-enhanced version of `getenv()`[6].
There are also some MFC string manipulations.

[5] http://msdn.microsoft.com/en-us/library/tb2sfw2z(VS.80).aspx
[6] Standard C library returning environment variable

Lots of other environment variables are checked as well. Here is a list of all variables being checked and what SAPGUI could write to trace log when logging is turned on:

DPTRACE	“GUI-OPTION: Trace set to %d”
TDW_HEXDUMP	“GUI-OPTION: Hexdump enabled”
TDW_WORKDIR	“GUI-OPTION: working directory ‘%s’”
TDW_SPLASHSRCEENOFF	“GUI-OPTION: Splash Screen Off” / “GUI-OPTION: Splash Screen On”
TDW_REPLYTIMEOUT	“GUI-OPTION: reply timeout %d milliseconds”
TDW_PLAYBACKTIMEOUT	“GUI-OPTION: PlaybackTimeout set to %d milliseconds”
TDW_NOCOMPRESS	“GUI-OPTION: no compression read”
TDW_EXPERT	“GUI-OPTION: expert mode”
TDW_PLAYBACKPROGRESS	“GUI-OPTION: PlaybackProgress”
TDW_PLAYBACKNETTRAFFIC	“GUI-OPTION: PlaybackNetTraffic”
TDW_PLAYLOG	“GUI-OPTION: /PlayLog is YES, file %s”
TDW_PLAYTIME	“GUI-OPTION: /PlayTime set to %d milliseconds”
TDW_LOGFILE	“GUI-OPTION: TDW_LOGFILE ‘%s’”
TDW_WAN	“GUI-OPTION: WAN - low speed connection enabled”
TDW_FULLMENU	“GUI-OPTION: FullMenu enabled”
SAP_CP / SAP_CODEPAGE	“GUI-OPTION: SAP_CODEPAGE ‘%d’”
UPDOWNLOAD_CP	“GUI-OPTION: UPDOWNLOAD_CP ‘%d’”
SNC_PARTNERNAME	“GUI-OPTION: SNC name ‘%s’”
SNC_QOP	“GUI-OPTION: SNC_QOP ‘%s’”
SNC_LIB	“GUI-OPTION: SNC is set to: %s”
SAPGUI_INPLACE	“GUI-OPTION: environment variable SAPGUI_INPLACE is on”

Settings for each variable are written to the array via pointer in `EDI` register. `EDI` is being set before that function call:

```
.text:6440EE00                lea     edi, [ebp+2884h+var_2884] ; options here like +0x15...
.text:6440EE03                lea     ecx, [esi+24h]
.text:6440EE06                call    load_command_line
.text:6440EE0B                mov     edi, eax
.text:6440EE0D                xor     ebx, ebx
.text:6440EE0F                cmp     edi, ebx
.text:6440EE11                jz      short loc_6440EE42
.text:6440EE13                push    edi
.text:6440EE14                push    offset aSapguiStoppedA ; "Sapgui stopped after commandline interp"...
.text:6440EE19                push    dword_644F93E8
.text:6440EE1F                call    FEWTraceError
```

Now, can we find *“data record mode switched on”* string? Yes, and here is the only reference in function `CDwsGui::Prepar`
How do I know class/method names? There is a lot of special debugging calls writing to log-files like:

```
.text:64405160                push    dword ptr [esi+2854h]
.text:64405166                push    offset aCdwsguiPrepare ; "\nCDwsGui::PrepareInfoWindow: sapgui env
    "...
.text:6440516B                push    dword ptr [esi+2848h]
.text:64405171                call    dbg
.text:64405176                add     esp, 0Ch
```

...or:

```
.text:6440237A                push    eax
.text:6440237B                push    offset aCclientStart_6 ; "CClient::Start: set shortcut user to
    '\%"...
.text:64402380                push    dword ptr [edi+4]
.text:64402383                call    dbg
.text:64402388                add     esp, 0Ch
```

It’s **very** useful.

So let’s see contents of that nagging pop-up window function:

```
.text:64404F4F CDwsGui__PrepareInfoWindow proc near
.text:64404F4F
.text:64404F4F pvParam         = byte ptr -3Ch
.text:64404F4F var_38          = dword ptr -38h
.text:64404F4F var_34          = dword ptr -34h
.text:64404F4F rc              = tagRECT ptr -2Ch
.text:64404F4F cy              = dword ptr -1Ch
.text:64404F4F h               = dword ptr -18h
.text:64404F4F var_14          = dword ptr -14h
.text:64404F4F var_10          = dword ptr -10h
.text:64404F4F var_4           = dword ptr -4
.text:64404F4F
.text:64404F4F                 push    30h
.text:64404F51                 mov     eax, offset loc_64438E00
.text:64404F56                 call    __EH_prolog3
.text:64404F5B                 mov     esi, ecx        ; ECX is pointer to object
.text:64404F5D                 xor     ebx, ebx
.text:64404F5F                 lea     ecx, [ebp+var_14]
.text:64404F62                 mov     [ebp+var_10], ebx

; demangled name: ATL::CStringT(void)
.text:64404F65                 call    ds:mfc90_316
.text:64404F6B                 mov     [ebp+var_4], ebx
.text:64404F6E                 lea     edi, [esi+2854h]
.text:64404F74                 push    offset aEnvironmentInf ; "Environment information:\n"
.text:64404F79                 mov     ecx, edi

; demangled name: ATL::CStringT::operator=(char const *)
.text:64404F7B                 call    ds:mfc90_820
.text:64404F81                 cmp     [esi+38h], ebx
.text:64404F84                 mov     ebx, ds:mfc90_2539
.text:64404F8A                 jbe     short loc_64404FA9
.text:64404F8C                 push    dword ptr [esi+34h]
.text:64404F8F                 lea     eax, [ebp+var_14]
.text:64404F92                 push    offset aWorkingDirecto ; "working directory: '\%s'\n"
.text:64404F97                 push    eax

; demangled name: ATL::CStringT::Format(char const *,...)
.text:64404F98                 call    ebx ; mfc90_2539
.text:64404F9A                 add     esp, 0Ch
.text:64404F9D                 lea     eax, [ebp+var_14]
.text:64404FA0                 push    eax
.text:64404FA1                 mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(class ATL::CSimpleStringT<char, 1> const &)
.text:64404FA3                 call    ds:mfc90_941
.text:64404FA9
.text:64404FA9 loc_64404FA9:
.text:64404FA9                 mov     eax, [esi+38h]
.text:64404FAC                 test    eax, eax
.text:64404FAE                 jbe     short loc_64404FD3
.text:64404FB0                 push    eax
.text:64404FB1                 lea     eax, [ebp+var_14]
.text:64404FB4                 push    offset aTraceLevelDAct ; "trace level \%d activated\n"
.text:64404FB9                 push    eax

; demangled name: ATL::CStringT::Format(char const *,...)
.text:64404FBA                 call    ebx ; mfc90_2539
.text:64404FBC                 add     esp, 0Ch
.text:64404FBF                 lea     eax, [ebp+var_14]
.text:64404FC2                 push    eax
.text:64404FC3                 mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(class ATL::CSimpleStringT<char, 1> const &)
.text:64404FC5                 call    ds:mfc90_941
```

```
.text:64404FCB                 xor     ebx, ebx
.text:64404FCD                 inc     ebx
.text:64404FCE                 mov     [ebp+var_10], ebx
.text:64404FD1                 jmp     short loc_64404FD6
.text:64404FD3 ; ---------------------------------------------------------------------------
.text:64404FD3
.text:64404FD3 loc_64404FD3:
.text:64404FD3                 xor     ebx, ebx
.text:64404FD5                 inc     ebx
.text:64404FD6
.text:64404FD6 loc_64404FD6:
.text:64404FD6                 cmp     [esi+38h], ebx
.text:64404FD9                 jbe     short loc_64404FF1
.text:64404FDB                 cmp     dword ptr [esi+2978h], 0
.text:64404FE2                 jz      short loc_64404FF1
.text:64404FE4                 push    offset aHexdumpInTrace ; "hexdump in trace activated\n"
.text:64404FE9                 mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64404FEB                 call    ds:mfc90_945
.text:64404FF1
.text:64404FF1 loc_64404FF1:
.text:64404FF1
.text:64404FF1                 cmp     byte ptr [esi+78h], 0
.text:64404FF5                 jz      short loc_64405007
.text:64404FF7                 push    offset aLoggingActivat ; "logging activated\n"
.text:64404FFC                 mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64404FFE                 call    ds:mfc90_945
.text:64405004                 mov     [ebp+var_10], ebx
.text:64405007
.text:64405007 loc_64405007:
.text:64405007                 cmp     byte ptr [esi+3Dh], 0
.text:6440500B                 jz      short bypass
.text:6440500D                 push    offset aDataCompressio ; "data compression switched off\n"
.text:64405012                 mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405014                 call    ds:mfc90_945
.text:6440501A                 mov     [ebp+var_10], ebx
.text:6440501D
.text:6440501D bypass:
.text:6440501D                 mov     eax, [esi+20h]
.text:64405020                 test    eax, eax
.text:64405022                 jz      short loc_6440503A
.text:64405024                 cmp     dword ptr [eax+28h], 0
.text:64405028                 jz      short loc_6440503A
.text:6440502A                 push    offset aDataRecordMode ; "data record mode switched on\n"
.text:6440502F                 mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405031                 call    ds:mfc90_945
.text:64405037                 mov     [ebp+var_10], ebx
.text:6440503A
.text:6440503A loc_6440503A:
.text:6440503A
.text:6440503A                 mov     ecx, edi
.text:6440503C                 cmp     [ebp+var_10], ebx
.text:6440503F                 jnz     loc_64405142
.text:64405045                 push    offset aForMaximumData ; "\nFor maximum data security delete\nthe s
    "...

; demangled name: ATL::CStringT::operator+=(char const *)
.text:6440504A                 call    ds:mfc90_945
.text:64405050                 xor     edi, edi
```

```
.text:64405052                 push    edi             ; fWinIni
.text:64405053                 lea     eax, [ebp+pvParam]
.text:64405056                 push    eax             ; pvParam
.text:64405057                 push    edi             ; uiParam
.text:64405058                 push    30h             ; uiAction
.text:6440505A                 call    ds:SystemParametersInfoA
.text:64405060                 mov     eax, [ebp+var_34]
.text:64405063                 cmp     eax, 1600
.text:64405068                 jle     short loc_64405072
.text:6440506A                 cdq
.text:6440506B                 sub     eax, edx
.text:6440506D                 sar     eax, 1
.text:6440506F                 mov     [ebp+var_34], eax
.text:64405072
.text:64405072 loc_64405072:
.text:64405072                 push    edi             ; hWnd
.text:64405073                 mov     [ebp+cy], 0A0h
.text:6440507A                 call    ds:GetDC
.text:64405080                 mov     [ebp+var_10], eax
.text:64405083                 mov     ebx, 12Ch
.text:64405088                 cmp     eax, edi
.text:6440508A                 jz      loc_64405113
.text:64405090                 push    11h             ; i
.text:64405092                 call    ds:GetStockObject
.text:64405098                 mov     edi, ds:SelectObject
.text:6440509E                 push    eax             ; h
.text:6440509F                 push    [ebp+var_10]    ; hdc
.text:644050A2                 call    edi ; SelectObject
.text:644050A4                 and     [ebp+rc.left], 0
.text:644050A8                 and     [ebp+rc.top], 0
.text:644050AC                 mov     [ebp+h], eax
.text:644050AF                 push    401h            ; format
.text:644050B4                 lea     eax, [ebp+rc]
.text:644050B7                 push    eax             ; lprc
.text:644050B8                 lea     ecx, [esi+2854h]
.text:644050BE                 mov     [ebp+rc.right], ebx
.text:644050C1                 mov     [ebp+rc.bottom], 0B4h

; demangled name: ATL::CSimpleStringT::GetLength(void)
.text:644050C8                 call    ds:mfc90_3178
.text:644050CE                 push    eax             ; cchText
.text:644050CF                 lea     ecx, [esi+2854h]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:644050D5                 call    ds:mfc90_910
.text:644050DB                 push    eax             ; lpchText
.text:644050DC                 push    [ebp+var_10]    ; hdc
.text:644050DF                 call    ds:DrawTextA
.text:644050E5                 push    4               ; nIndex
.text:644050E7                 call    ds:GetSystemMetrics
.text:644050ED                 mov     ecx, [ebp+rc.bottom]
.text:644050F0                 sub     ecx, [ebp+rc.top]
.text:644050F3                 cmp     [ebp+h], 0
.text:644050F7                 lea     eax, [eax+ecx+28h]
.text:644050FB                 mov     [ebp+cy], eax
.text:644050FE                 jz      short loc_64405108
.text:64405100                 push    [ebp+h]         ; h
.text:64405103                 push    [ebp+var_10]    ; hdc
.text:64405106                 call    edi ; SelectObject
.text:64405108
.text:64405108 loc_64405108:
.text:64405108                 push    [ebp+var_10]    ; hDC
.text:6440510B                 push    0               ; hWnd
.text:6440510D                 call    ds:ReleaseDC
.text:64405113
.text:64405113 loc_64405113:
```

```
.text:64405113                 mov     eax, [ebp+var_38]
.text:64405116                 push    80h             ; uFlags
.text:6440511B                 push    [ebp+cy]        ; cy
.text:6440511E                 inc     eax
.text:6440511F                 push    ebx             ; cx
.text:64405120                 push    eax             ; Y
.text:64405121                 mov     eax, [ebp+var_34]
.text:64405124                 add     eax, 0FFFFFED4h
.text:64405129                 cdq
.text:6440512A                 sub     eax, edx
.text:6440512C                 sar     eax, 1
.text:6440512E                 push    eax             ; X
.text:6440512F                 push    0               ; hWndInsertAfter
.text:64405131                 push    dword ptr [esi+285Ch] ; hWnd
.text:64405137                 call    ds:SetWindowPos
.text:6440513D                 xor     ebx, ebx
.text:6440513F                 inc     ebx
.text:64405140                 jmp     short loc_6440514D
.text:64405142 ; ---------------------------------------------------------------------------
.text:64405142
.text:64405142 loc_64405142:
.text:64405142                 push    offset byte_64443AF8

; demangled name: ATL::CStringT::operator=(char const *)
.text:64405147                 call    ds:mfc90_820
.text:6440514D
.text:6440514D loc_6440514D:
.text:6440514D                 cmp     dword_6450B970, ebx
.text:64405153                 jl      short loc_64405188
.text:64405155                 call    sub_6441C910
.text:6440515A                 mov     dword_644F858C, ebx
.text:64405160                 push    dword ptr [esi+2854h]
.text:64405166                 push    offset aCdwsguiPrepare ; "\nCDwsGui::PrepareInfoWindow: sapgui env
    "...
.text:6440516B                 push    dword ptr [esi+2848h]
.text:64405171                 call    dbg
.text:64405176                 add     esp, 0Ch
.text:64405179                 mov     dword_644F858C, 2
.text:64405183                 call    sub_6441C920
.text:64405188
.text:64405188 loc_64405188:
.text:64405188                 or      [ebp+var_4], 0FFFFFFFFh
.text:6440518C                 lea     ecx, [ebp+var_14]

; demangled name: ATL::CStringT::~CStringT()
.text:6440518F                 call    ds:mfc90_601
.text:64405195                 call    __EH_epilog3
.text:6440519A                 retn
.text:6440519A CDwsGui__PrepareInfoWindow endp
```

ECX at function start gets pointer to object (because it is thiscall 2.5.4-type of function). In our case, that object obviously has class type *CDwsGui*. Depends of option turned on in the object, specific message part will be concatenated to resulting message.

If value at this+0x3D address is not zero, compression is off:

```
.text:64405007 loc_64405007:
.text:64405007                 cmp     byte ptr [esi+3Dh], 0
.text:6440500B                 jz      short bypass
.text:6440500D                 push    offset aDataCompressio ; "data compression switched off\n"
.text:64405012                 mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405014                 call    ds:mfc90_945
.text:6440501A                 mov     [ebp+var_10], ebx
.text:6440501D
.text:6440501D bypass:
```

It is interesting, that finally, *var_10* variable state defines whether the message is to be shown at all:

```
.text:6440503C                cmp     [ebp+var_10], ebx
.text:6440503F                jnz     exit ; bypass drawing

; add strings "For maximum data security delete" / "the setting(s) as soon as possible !":

.text:64405045                push    offset aForMaximumData ; "\nFor maximum data security delete\nthe s
    "...
.text:6440504A                call    ds:mfc90_945 ; ATL::CStringT::operator+=(char const *)
.text:64405050                xor     edi, edi
.text:64405052                push    edi             ; fWinIni
.text:64405053                lea     eax, [ebp+pvParam]
.text:64405056                push    eax             ; pvParam
.text:64405057                push    edi             ; uiParam
.text:64405058                push    30h             ; uiAction
.text:6440505A                call    ds:SystemParametersInfoA
.text:64405060                mov     eax, [ebp+var_34]
.text:64405063                cmp     eax, 1600
.text:64405068                jle     short loc_64405072
.text:6440506A                cdq
.text:6440506B                sub     eax, edx
.text:6440506D                sar     eax, 1
.text:6440506F                mov     [ebp+var_34], eax
.text:64405072
.text:64405072 loc_64405072:

start drawing:

.text:64405072                push    edi             ; hWnd
.text:64405073                mov     [ebp+cy], 0A0h
.text:6440507A                call    ds:GetDC
```

Let's check our theory on practice.

JNZ at this line ...

```
.text:6440503F                jnz     exit ; bypass drawing
```

...replace it with just JMP, and get SAPGUI working without that nagging pop-up window appearing!

Now let's dig deeper and find connection between 0x15 offset in `load_command_line()` (I gave that name to that function) and `this`+0x3D variable in *CDwsGui::PrepareInfoWindow*. Are we sure that the value is the same?

I'm starting to search for all occurences of 0x15 value in code. For some small programs like SAPGUI, it sometimes works. Here is the first occurence I got:

```
.text:64404C19 sub_64404C19    proc near
.text:64404C19
.text:64404C19 arg_0           = dword ptr  4
.text:64404C19
.text:64404C19                 push    ebx
.text:64404C1A                 push    ebp
.text:64404C1B                 push    esi
.text:64404C1C                 push    edi
.text:64404C1D                 mov     edi, [esp+10h+arg_0]
.text:64404C21                 mov     eax, [edi]
.text:64404C23                 mov     esi, ecx ; ESI/ECX are pointers to some unknown object.
.text:64404C25                 mov     [esi], eax
.text:64404C27                 mov     eax, [edi+4]
.text:64404C2A                 mov     [esi+4], eax
.text:64404C2D                 mov     eax, [edi+8]
.text:64404C30                 mov     [esi+8], eax
.text:64404C33                 lea     eax, [edi+0Ch]
.text:64404C36                 push    eax
.text:64404C37                 lea     ecx, [esi+0Ch]

; demangled name:  ATL::CStringT::operator=(class ATL::CStringT ... &)
.text:64404C3A                 call    ds:mfc90_817
```

```
.text:64404C40                mov     eax, [edi+10h]
.text:64404C43                mov     [esi+10h], eax
.text:64404C46                mov     al, [edi+14h]
.text:64404C49                mov     [esi+14h], al
.text:64404C4C                mov     al, [edi+15h] ; copy byte from 0x15 offset
.text:64404C4F                mov     [esi+15h], al ; to 0x15 offset in CDwsGui object
```

That function was called from the function named *CDwsGui::CopyOptions*! And thanks again for debugging information.

But the real answer in the function *CDwsGui::Init()*:

```
.text:6440B0BF loc_6440B0BF:
.text:6440B0BF                mov     eax, [ebp+arg_0]
.text:6440B0C2                push    [ebp+arg_4]
.text:6440B0C5                mov     [esi+2844h], eax
.text:6440B0CB                lea     eax, [esi+28h] ; ESI is pointer to CDwsGui object
.text:6440B0CE                push    eax
.text:6440B0CF                call    CDwsGui__CopyOptions
```

Finally, we understand: array filled in `load_command_line()` is actually placed in *CDwsGui* class but on `this`+0x28 address. 0x15 + 0x28 is exactly 0x3D. OK, we found the place where the value is copied to.

Let's also find other places where 0x3D offset is used. Here is one of them in *CDwsGui::SapguiRun* function (again, thanks to debugging calls):

```
.text:64409D58                cmp     [esi+3Dh], bl   ; ESI is pointer to CDwsGui object
.text:64409D5B                lea     ecx, [esi+2B8h]
.text:64409D61                setz    al
.text:64409D64                push    eax             ; arg_10 of CConnectionContext::CreateNetwork
.text:64409D65                push    dword ptr [esi+64h]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:64409D68                call    ds:mfc90_910
.text:64409D68                                        ; no arguments
.text:64409D6E                push    eax
.text:64409D6F                lea     ecx, [esi+2BCh]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:64409D75                call    ds:mfc90_910
.text:64409D75                                        ; no arguments
.text:64409D7B                push    eax
.text:64409D7C                push    esi
.text:64409D7D                lea     ecx, [esi+8]
.text:64409D80                call    CConnectionContext__CreateNetwork
```

Let's check our findings. Replace `setz al` here to `xor eax, eax / nop`, clear TDW_NOCOMPRESS environment variable and run SAPGUI. Wow! There is no nagging window (just as expected, because variable is not set), but in Wireshark we can see that the network packets are not compressed anymore! Obviously, this is the place where compression flag is to be set in *CConnectionContext* object.

So, compression flag is passed in the 5th argument of function *CConnectionContext::CreateNetwork*. Inside that function, another one is called:

```
...
.text:64403476                push    [ebp+compression]
.text:64403479                push    [ebp+arg_C]
.text:6440347C                push    [ebp+arg_8]
.text:6440347F                push    [ebp+arg_4]
.text:64403482                push    [ebp+arg_0]
.text:64403485                call    CNetwork__CNetwork
```

Compression flag is passing here in the 5th argument to *CNetwork::CNetwork* constructor.

And here is how *CNetwork* constructor sets some flag in *CNetwork* object according to the 5th argument **and** some another variable which probably could affect network packets compression too.

```
.text:64411DF1                cmp     [ebp+compression], esi
.text:64411DF7                jz      short set_EAX_to_0
```

```
.text:64411DF9                mov     al, [ebx+78h]   ; another value may affect compression?
.text:64411DFC                cmp     al, '3'
.text:64411DFE                jz      short set_EAX_to_1
.text:64411E00                cmp     al, '4'
.text:64411E02                jnz     short set_EAX_to_0
.text:64411E04
.text:64411E04 set_EAX_to_1:
.text:64411E04                xor     eax, eax
.text:64411E06                inc     eax             ; EAX -> 1
.text:64411E07                jmp     short loc_64411E0B
.text:64411E09 ; ---------------------------------------------------------------------------
.text:64411E09
.text:64411E09 set_EAX_to_0:
.text:64411E09
.text:64411E09                xor     eax, eax        ; EAX -> 0
.text:64411E0B
.text:64411E0B loc_64411E0B:
.text:64411E0B                mov     [ebx+3A4h], eax ; EBX is pointer to CNetwork object
```

At this point we know that compression flag is stored in *CNetwork* class at *this+0x3A4* address.

Now let's dig across SAPguilib.dll for 0x3A4 value. And here is the second occurence in *CDwsGui::OnClientMessageWrite* (endless thanks for debugging information):

```
.text:64406F76 loc_64406F76:
.text:64406F76                mov     ecx, [ebp+7728h+var_7794]
.text:64406F79                cmp     dword ptr [ecx+3A4h], 1
.text:64406F80                jnz     compression_flag_is_zero
.text:64406F86                mov     byte ptr [ebx+7], 1
.text:64406F8A                mov     eax, [esi+18h]
.text:64406F8D                mov     ecx, eax
.text:64406F8F                test    eax, eax
.text:64406F91                ja      short loc_64406FFF
.text:64406F93                mov     ecx, [esi+14h]
.text:64406F96                mov     eax, [esi+20h]
.text:64406F99
.text:64406F99 loc_64406F99:
.text:64406F99                push    dword ptr [edi+2868h] ; int
.text:64406F9F                lea     edx, [ebp+7728h+var_77A4]
.text:64406FA2                push    edx             ; int
.text:64406FA3                push    30000           ; int
.text:64406FA8                lea     edx, [ebp+7728h+Dst]
.text:64406FAB                push    edx             ; Dst
.text:64406FAC                push    ecx             ; int
.text:64406FAD                push    eax             ; Src
.text:64406FAE                push    dword ptr [edi+28C0h] ; int
.text:64406FB4                call    sub_644055C5       ; actual compression routine
.text:64406FB9                add     esp, 1Ch
.text:64406FBC                cmp     eax, 0FFFFFFF6h
.text:64406FBF                jz      short loc_64407004
.text:64406FC1                cmp     eax, 1
.text:64406FC4                jz      loc_6440708C
.text:64406FCA                cmp     eax, 2
.text:64406FCD                jz      short loc_64407004
.text:64406FCF                push    eax
.text:64406FD0                push    offset aCompressionErr ; "compression error [rc = \%d]- program wi
    "...
.text:64406FD5                push    offset aGui_err_compre ; "GUI_ERR_COMPRESS"
.text:64406FDA                push    dword ptr [edi+28D0h]
.text:64406FE0                call    SapPcTxtRead
```

Let's take a look into *sub_644055C5*. In it we can only see call to memcpy() and some other function named (by IDA 5) *sub_64417440*.

And, let's take a look inside *sub_64417440*. What we see is:

```
.text:6441747C                push    offset aErrorCsrcompre ; "\nERROR: CsRCompress: invalid handle"
.text:64417481                call    eax ; dword_644F94C8
```

```
.text:64417483                add     esp, 4
```

Voilà! We've found the function which actually compresses data. As I revealed in past, this function is used in SAP and also open-source MaxDB project. So it is available in sources.

Doing last check here:

```
.text:64406F79                cmp     dword ptr [ecx+3A4h], 1
.text:64406F80                jnz     compression_flag_is_zero
```

Replace JNZ here for unconditional JMP. Remove environment variable TDW_NOCOMPRESS. Voilà! In Wireshark we see that client messages are not compressed. Server responses, however, are compressed.

So we found exact connection between environment variable and the point where data compression routine may be called or may be bypassed.

7.2.2 SAP 6.0 password checking functions

While returning again to my SAP 6.0 IDES installed in VMware box, I figured out I forgot the password for SAP* account, then it back to my memory, but now I got error message «*Password logon no longer possible - too many failed attempts*», because I've spent all these attempts in trying to recall it.

First extremely good news is that full *disp+work.pdb* file is supplied with SAP, it contain almost everything: function names, structures, types, local variable and argument names, etc. What a lavish gift!

I got TYPEINFODUMP[7] utility for converting PDB files into something readable and grepable.

Here is an example of function information + its arguments + its local variables:

```
FUNCTION ThVmcSysEvent
  Address:         10143190  Size:      675 bytes  Index:    60483  TypeIndex:    60484
  Type: int NEAR_C ThVmcSysEvent (unsigned int, unsigned char, unsigned short*)
Flags: 0
PARAMETER events
  Address: Reg335+288  Size:        4 bytes  Index:    60488  TypeIndex:    60489
  Type: unsigned int
Flags: d0
PARAMETER opcode
  Address: Reg335+296  Size:        1 bytes  Index:    60490  TypeIndex:    60491
  Type: unsigned char
Flags: d0
PARAMETER serverName
  Address: Reg335+304  Size:        8 bytes  Index:    60492  TypeIndex:    60493
  Type: unsigned short*
Flags: d0
STATIC_LOCAL_VAR func
  Address:         12274af0  Size:        8 bytes  Index:    60495  TypeIndex:    60496
  Type: wchar_t*
Flags: 80
LOCAL_VAR admhead
  Address: Reg335+304  Size:        8 bytes  Index:    60498  TypeIndex:    60499
  Type: unsigned char*
Flags: 90
LOCAL_VAR record
  Address: Reg335+64  Size:      204 bytes  Index:    60501  TypeIndex:    60502
  Type: AD_RECORD
Flags: 90
LOCAL_VAR adlen
  Address: Reg335+296  Size:        4 bytes  Index:    60508  TypeIndex:    60509
  Type: int
Flags: 90
```

And here is an example of some structure:

```
STRUCT DBSL_STMTID
Size: 120  Variables: 4  Functions: 0  Base classes: 0
MEMBER moduletype
```

[7] http://www.debuginfo.com/tools/typeinfodump.html

```
  Type:  DBSL_MODULETYPE
  Offset:        0  Index:        3  TypeIndex:    38653
MEMBER module
  Type:  wchar_t module[40]
  Offset:        4  Index:        3  TypeIndex:      831
MEMBER stmtnum
  Type:  long
  Offset:       84  Index:        3  TypeIndex:      440
MEMBER timestamp
  Type:  wchar_t timestamp[15]
  Offset:       88  Index:        3  TypeIndex:     6612
```

Wow!

Another good news is: *debugging* calls (there are plenty of them) are very useful.

Here you can also notice *ct_level* global variable[8], reflecting current trace level.

There is a lot of such debugging inclusions in *disp+work.exe*:

```
cmp     cs:ct_level, 1
jl      short loc_1400375DA
call    DpLock
lea     rcx, aDpxxtool4_c ; "dpxxtool4.c"
mov     edx, 4Eh        ; line
call    CTrcSaveLocation
mov     r8, cs:func_48
mov     rcx, cs:hdl     ; hdl
lea     rdx, aSDpreadmemvalu ; "%s: DpReadMemValue (%d)"
mov     r9d, ebx
call    DpTrcErr
call    DpUnlock
```

If current trace level is bigger or equal to threshold defined in the code here, debugging message will be written to log files like *dev_w0*, *dev_disp*, and other *dev** files.

Let's do grepping on file I got with the help of TYPEINFODUMP utility:

```
cat "disp+work.pdb.d" | grep FUNCTION | grep -i password
```

I got:

```
FUNCTION rcui::AgiPassword::DiagISelection
FUNCTION ssf_password_encrypt
FUNCTION ssf_password_decrypt
FUNCTION password_logon_disabled
FUNCTION dySignSkipUserPassword
FUNCTION migrate_password_history
FUNCTION password_is_initial
FUNCTION rcui::AgiPassword::IsVisible
FUNCTION password_distance_ok
FUNCTION get_password_downwards_compatibility
FUNCTION dySignUnSkipUserPassword
FUNCTION rcui::AgiPassword::GetTypeName
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$2
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$0
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$1
FUNCTION usm_set_password
FUNCTION rcui::AgiPassword::TraceTo
FUNCTION days_since_last_password_change
FUNCTION rsecgrp_generate_random_password
FUNCTION rcui::AgiPassword::`scalar deleting destructor'
FUNCTION password_attempt_limit_exceeded
FUNCTION handle_incorrect_password
FUNCTION `rcui::AgiPassword::`scalar deleting destructor''::`1'::dtor$1
FUNCTION calculate_new_password_hash
FUNCTION shift_password_to_history
```

[8] More about trace level: http://help.sap.com/saphelp_nwpi71/helpdata/en/46/962416a5a613e8e10000000a155369/content.htm

```
FUNCTION rcui::AgiPassword::GetType
FUNCTION found_password_in_history
FUNCTION `rcui::AgiPassword::`scalar deleting destructor''::`1'::dtor$0
FUNCTION rcui::AgiObj::IsaPassword
FUNCTION password_idle_check
FUNCTION SlicHwPasswordForDay
FUNCTION rcui::AgiPassword::IsaPassword
FUNCTION rcui::AgiPassword::AgiPassword
FUNCTION delete_user_password
FUNCTION usm_set_user_password
FUNCTION Password_API
FUNCTION get_password_change_for_SSO
FUNCTION password_in_USR40
FUNCTION rsec_agrp_abap_generate_random_password
```

Let's also try to search for debug messages which contain words *«password»* and *«locked»*. One of them is the string *«user was locked by subsequently failed password logon attempts»* referenced in function *password_attempt_limit_exceeded()*.

Other string this function I found may write to log file are: *«password logon attempt will be rejected immediately (preventing dictionary attacks)»*, *«failed-logon lock: expired (but not removed due to 'read-only' operation)»*, *«failed-logon lock: expired => removed»*.

After playing for a little with this function, I quickly noticed that the problem is exactly in it. It is called from *chckpass()* function — one of the password checking functions.

First, I would like to be sure I'm at the correct point:

Run my *tracer* 5.0.1:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!chckpass,args:3,unicode
```

```
PID=2236|TID=2248|(0) disp+work.exe!chckpass (0x202c770, L"Brewered1                                ", 0x41)
    (called from 0x1402f1060 (disp+work.exe!usrexist+0x3c0))
PID=2236|TID=2248|(0) disp+work.exe!chckpass -> 0x35
```

Call path is: *syssigni() -> DyISigni() -> dychkusr() -> usrexist() -> chckpass()*.

Number 0x35 is error returning in *chckpass()* at that point:

```
.text:00000001402ED567 loc_1402ED567:                          ; CODE XREF: chckpass+B4
.text:00000001402ED567                 mov     rcx, rbx        ; usr02
.text:00000001402ED56A                 call    password_idle_check
.text:00000001402ED56F                 cmp     eax, 33h
.text:00000001402ED572                 jz      loc_1402EDB4E
.text:00000001402ED578                 cmp     eax, 36h
.text:00000001402ED57B                 jz      loc_1402EDB3D
.text:00000001402ED581                 xor     edx, edx        ; usr02_readonly
.text:00000001402ED583                 mov     rcx, rbx        ; usr02
.text:00000001402ED586                 call    password_attempt_limit_exceeded
.text:00000001402ED58B                 test    al, al
.text:00000001402ED58D                 jz      short loc_1402ED5A0
.text:00000001402ED58F                 mov     eax, 35h
.text:00000001402ED594                 add     rsp, 60h
.text:00000001402ED598                 pop     r14
.text:00000001402ED59A                 pop     r12
.text:00000001402ED59C                 pop     rdi
.text:00000001402ED59D                 pop     rsi
.text:00000001402ED59E                 pop     rbx
.text:00000001402ED59F                 retn
```

Fine, let's check:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!password_attempt_limit_exceeded,args:4,unicode,rt:0
```

```
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded (0x202c770, 0, 0x257758, 0) (called from
    0x1402ed58b (disp+work.exe!chckpass+0xeb))
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded -> 1
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
```

```
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded (0x202c770, 0, 0, 0) (called from 0
    x1402e9794 (disp+work.exe!chngpass+0xe4))
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded -> 1
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
```

Excellent! I can successfully login now.

By the way, if I try to pretend I forgot the password, fixing *chckpass()* function return value at zero is enough to bypass check:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!chckpass,args:3,unicode,rt:0
```

```
PID=2744|TID=360|(0) disp+work.exe!chckpass (0x202c770, L"bogus                                    ", 0x41) (
    called from 0x1402f1060 (disp+work.exe!usrexist+0x3c0))
PID=2744|TID=360|(0) disp+work.exe!chckpass -> 0x35
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
```

What also can be said while analyzing *password_attempt_limit_exceeded()* function is that at the very beginning of it, this call might be seen:

```
lea     rcx, aLoginFailed_us ; "login/failed_user_auto_unlock"
call    sapgparam
test    rax, rax
jz      short loc_1402E19DE
movzx   eax, word ptr [rax]
cmp     ax, 'N'
jz      short loc_1402E19D4
cmp     ax, 'n'
jz      short loc_1402E19D4
cmp     ax, '0'
jnz     short loc_1402E19DE
```

Obviously, function *sapgparam()* used to query value of some configuration parameter. This function can be called from 1768 different places. It seems, with the help of this information, we can easily find places in code, control flow of which can be affected by specific configuration parameters.

It is really sweet. Function names are very clear, much clearer than in Oracle RDBMS. It seems, *disp+work* process written in C++. It was apparently rewritten some time ago?

7.3 Oracle RDBMS

7.3.1 `V$VERSION` table in Oracle RDBMS

Oracle RDBMS 11.2 is a huge program, main module `oracle.exe` contain approx. 124,000 functions. For comparison, Windows 7 x86 kernel (ntoskrnl.exe) — approx. 11,000 functions and Linux 3.9.8 kernel (with default drivers compiled) — 31,000 functions.

Let's start with an easy question. Where Oracle RDBMS get all this information, when we execute such simple statement in SQL*Plus:

```
SQL> select * from V$VERSION;
```

And we've got:

```
BANNER
--------------------------------------------------------------------------------

Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
PL/SQL Release 11.2.0.1.0 - Production
CORE    11.2.0.1.0      Production
TNS for 32-bit Windows: Version 11.2.0.1.0 - Production
NLSRTL Version 11.2.0.1.0 - Production
```

Let's start. Where in Oracle RDBMS we may find a string `V$VERSION`?

As of win32-version, `oracle.exe` file contain that string, that can be investigated easily. But we can also use object (.o) files from Linux version of Oracle RDBMS, because, unlike win32 version `oracle.exe`, function names (and global variables as well) are preserved there.

So, `kqf.o` file contain `V$VERSION` string. That object file is in main Oracle-library `libserver11.a`.

A reference to this text string we may find in `kqfviw` table stored in the same file, `kqf.o`:

Listing 7.1: kqf.o

```
.rodata:0800C4A0 kqfviw          dd 0Bh                  ; DATA XREF: kqfchk:loc_8003A6D
.rodata:0800C4A0                                         ; kqfgbn+34
.rodata:0800C4A4                 dd offset _2__STRING_10102_0 ; "GV$WAITSTAT"
.rodata:0800C4A8                 dd 4
.rodata:0800C4AC                 dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C4B0                 dd 3
.rodata:0800C4B4                 dd 0
.rodata:0800C4B8                 dd 195h
.rodata:0800C4BC                 dd 4
.rodata:0800C4C0                 dd 0
.rodata:0800C4C4                 dd 0FFFFFC1CBh
.rodata:0800C4C8                 dd 3
.rodata:0800C4CC                 dd 0
.rodata:0800C4D0                 dd 0Ah
.rodata:0800C4D4                 dd offset _2__STRING_10104_0 ; "V$WAITSTAT"
.rodata:0800C4D8                 dd 4
.rodata:0800C4DC                 dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C4E0                 dd 3
.rodata:0800C4E4                 dd 0
.rodata:0800C4E8                 dd 4Eh
.rodata:0800C4EC                 dd 3
.rodata:0800C4F0                 dd 0
.rodata:0800C4F4                 dd 0FFFFC003h
.rodata:0800C4F8                 dd 4
.rodata:0800C4FC                 dd 0
.rodata:0800C500                 dd 5
.rodata:0800C504                 dd offset _2__STRING_10105_0 ; "GV$BH"
.rodata:0800C508                 dd 4
.rodata:0800C50C                 dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C510                 dd 3
.rodata:0800C514                 dd 0
.rodata:0800C518                 dd 269h
.rodata:0800C51C                 dd 15h
.rodata:0800C520                 dd 0
.rodata:0800C524                 dd 0FFFFC1EDh
.rodata:0800C528                 dd 8
.rodata:0800C52C                 dd 0
.rodata:0800C530                 dd 4
.rodata:0800C534                 dd offset _2__STRING_10106_0 ; "V$BH"
.rodata:0800C538                 dd 4
.rodata:0800C53C                 dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C540                 dd 3
.rodata:0800C544                 dd 0
.rodata:0800C548                 dd 0F5h
.rodata:0800C54C                 dd 14h
.rodata:0800C550                 dd 0
.rodata:0800C554                 dd 0FFFFC1EEh
.rodata:0800C558                 dd 5
.rodata:0800C55C                 dd 0
```

By the way, often, while analysing Oracle RDBMS internals, you may ask yourself, why functions and global variable names are so weird. Probably, because Oracle RDBMS is very old product and was developed in C in 1980-s. And that was a time when C standard guaranteed function names/variables support only up to 6 characters inclusive: «6 significant initial characters in an external identifier»[9]

Probably, the table `kqfviw` contain most (maybe even all) views prefixed with V$, these are *fixed views*, present

[9] Draft ANSI C Standard (ANSI X3J11/88-090) (May 13, 1988)

all the time. Superficially, by noticing cyclic recurrence of data, we can easily see that each `kqfviw` table element has 12 32-bit fields. It's very simple to create a 12-elements structure in IDA 5 and apply it to all table elements. As of Oracle RDBMS version 11.2, there are 1023 table elements, i.e., there are described 1023 of all possible *fixed views*. We will return to this number later.

As we can see, there are not much information in these numbers in fields. The very first number is always equals to name of view (without terminating zero. This is correct for each element. But this information is not very useful.

We also know that information about all fixed views can be retrieved from *fixed view* named `V$FIXED_VIEW_DEFINITION` (by the way, the information for this view is also taken from `kqfviw` and `kqfvip` tables.) By the way, there are 1023 elemens too.

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='V$VERSION';

VIEW_NAME
------------------------------
VIEW_DEFINITION
--------------------------------------------------------------------------------

V$VERSION
select  BANNER from GV$VERSION where inst_id = USERENV('Instance')
```

So, `V$VERSION` is some kind of *thunk view* for another view, named `GV$VERSION`, which is, in turn:

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='GV$VERSION';

VIEW_NAME
------------------------------
VIEW_DEFINITION
--------------------------------------------------------------------------------

GV$VERSION
select inst_id, banner from x$version
```

Tables prefixed as X$ in Oracle RDBMS– is service tables too, undocumented, cannot be changed by user and refreshed dynamically.

Let's also try to search the text `select BANNER from GV$VERSION where inst_id = USERENV('Instance')` in `kqf.o` file and we find it in `kqfvip` table:

.

Listing 7.2: kqf.o

```
rodata:080185A0 kqfvip          dd offset _2__STRING_11126_0 ; DATA XREF: kqfgvcn+18
.rodata:080185A0                                             ; kqfgvt+F
.rodata:080185A0                                             ; "select inst_id,decode(indx,1,'data bloc"...
.rodata:080185A4                 dd offset kqfv459_c_0
.rodata:080185A8                 dd 0
.rodata:080185AC                 dd 0

...

.rodata:08019570                 dd offset _2__STRING_11378_0 ; "select  BANNER from GV$VERSION where in"...
.rodata:08019574                 dd offset kqfv133_c_0
.rodata:08019578                 dd 0
.rodata:0801957C                 dd 0
.rodata:08019580                 dd offset _2__STRING_11379_0 ; "select inst_id,decode(bitand(cfflg,1),0"...
.rodata:08019584                 dd offset kqfv403_c_0
.rodata:08019588                 dd 0
.rodata:0801958C                 dd 0
.rodata:08019590                 dd offset _2__STRING_11380_0 ; "select  STATUS , NAME, IS_RECOVERY_DEST"...
.rodata:08019594                 dd offset kqfv199_c_0
```

The table appear to have 4 fields in each element. By the way, there are 1023 elements too. The second field pointing to another table, containing table fields for this *fixed view*. As of `V$VERSION`, this table contain only two elements, first is 6 and second is `BANNER` string (the number (6) is this string length) and after, *terminating* element contain 0 and *null* C-string:

Listing 7.3: kqf.o

```
.rodata:080BBAC4 kqfv133_c_0     dd 6                    ; DATA XREF: .rodata:08019574
.rodata:080BBAC8                 dd offset _2__STRING_5017_0 ; "BANNER"
.rodata:080BBACC                 dd 0
.rodata:080BBAD0                 dd offset _2__STRING_0_0
```

By joining data from both `kqfviw` and `kqfvip` tables, we may get SQL-statements which are executed when user wants to query information from specific *fixed view*.

So I wrote an oracle tables[10] program, so to gather all this information from Oracle RDBMS for Linux object files. For `V$VERSION`, we may find this:

Listing 7.4: Result of oracle tables

```
kqfviw_element.viewname: [V$VERSION] ?: 0x3 0x43 0x1 0xffffc085 0x4
kqfvip_element.statement: [select  BANNER from GV$VERSION where inst_id = USERENV('Instance')]
kqfvip_element.params:
[BANNER]
```

And:

Listing 7.5: Result of oracle tables

```
kqfviw_element.viewname: [GV$VERSION] ?: 0x3 0x26 0x2 0xffffc192 0x1
kqfvip_element.statement: [select inst_id, banner from x$version]
kqfvip_element.params:
[INST_ID] [BANNER]
```

`GV$VERSION` *fixed view* is distinct from `V$VERSION` in only that way that it contains one more field with *instance* identifier. Anyway, we stuck at the table `X$VERSION`. Just like any other X$-tables, it's undocumented, however, we can query it:

```
SQL> select * from x$version;

ADDR           INDX    INST_ID
-------- ---------- ----------
BANNER
--------------------------------------------------------------------------------

0DBAF574          0          1
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production

...
```

This table has additional fields like `ADDR` and `INDX`.

While scrolling `kqf.o` in IDA 5 we may spot another table containing pointer to `X$VERSION` string, this is `kqftab`:

Listing 7.6: kqf.o

```
.rodata:0803CAC0                 dd 9                    ; element number 0x1f6
.rodata:0803CAC4                 dd offset _2__STRING_13113_0 ; "X$VERSION"
.rodata:0803CAC8                 dd 4
.rodata:0803CACC                 dd offset _2__STRING_13114_0 ; "kqvt"
.rodata:0803CAD0                 dd 4
.rodata:0803CAD4                 dd 4
.rodata:0803CAD8                 dd 0
.rodata:0803CADC                 dd 4
.rodata:0803CAE0                 dd 0Ch
.rodata:0803CAE4                 dd 0FFFFC075h
.rodata:0803CAE8                 dd 3
.rodata:0803CAEC                 dd 0
.rodata:0803CAF0                 dd 7
.rodata:0803CAF4                 dd offset _2__STRING_13115_0 ; "X$KQFSZ"
.rodata:0803CAF8                 dd 5
.rodata:0803CAFC                 dd offset _2__STRING_13116_0 ; "kqfsz"
.rodata:0803CB00                 dd 1
```

[10] http://yurichev.com/oracle_tables.html

```
.rodata:0803CB04                dd 38h
.rodata:0803CB08                dd 0
.rodata:0803CB0C                dd 7
.rodata:0803CB10                dd 0
.rodata:0803CB14                dd 0FFFFC09Dh
.rodata:0803CB18                dd 2
.rodata:0803CB1C                dd 0
```

There are a lot of references to X$-table names, apparently, to all Oracle RDBMS 11.2 X$-tables. But again, we have not enough information. I have no idea, what `kqvt` string mean. `kq` prefix may mean *kernal* and *query*. `v`, apparently, mean *version* and `t` — *type*? Frankly speaking, I don't know.

The table named similarly can be found in `kqf.o`:

Listing 7.7: kqf.o

```
.rodata:0808C360 kqvt_c_0       kqftap_param <4, offset _2__STRING_19_0, 917h, 0, 0, 0, 4, 0, 0>
.rodata:0808C360                                        ; DATA XREF: .rodata:08042680
.rodata:0808C360                                        ; "ADDR"
.rodata:0808C384                kqftap_param <4, offset _2__STRING_20_0, 0B02h, 0, 0, 0, 4, 0, 0> ; "INDX"
.rodata:0808C3A8                kqftap_param <7, offset _2__STRING_21_0, 0B02h, 0, 0, 0, 4, 0, 0> ; "
    INST_ID"
.rodata:0808C3CC                kqftap_param <6, offset _2__STRING_5017_0, 601h, 0, 0, 0, 50h, 0, 0> ; "
    BANNER"
.rodata:0808C3F0                kqftap_param <0, offset _2__STRING_0_0, 0, 0, 0, 0, 0, 0, 0>
```

It contain information about all fields in the `X$VERSION` table. The only reference to this table present in `kqftap` table:

Listing 7.8: kqf.o

```
.rodata:08042680                kqftap_element <0, offset kqvt_c_0, offset kqvrow, 0> ; element 0x1f6
```

It's interesting that this element here is $0x1f6th$ (502nd), just as a pointer to `X$VERSION` string in `kqftab` table. Probably, `kqftap` and `kqftab` tables are complement each other, just like `kqfvip` and `kqfviw`. We also see a pointer to `kqvrow()` function. Something useful at last!

So I added these tables to my oracle tables[11] utility too. For `X$VERSION` I've got:

Listing 7.9: Result of oracle tables

```
kqftab_element.name: [X$VERSION] ?: [kqvt] 0x4 0x4 0x4 0xc 0xffffc075 0x3
kqftap_param.name=[ADDR] ?: 0x917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[BANNER] ?: 0x601 0x0 0x0 0x0 0x50 0x0 0x0
kqftap_element.fn1=kqvrow
kqftap_element.fn2=NULL
```

Using *tracer* 5.0.1, it's easy to check that this function called 6 times in row (from the `qerfxFetch()` function) while querying `X$VERSION` table.

Let's run *tracer* 5.0.1 in `cc` mode (it will comment each executed instruction):

```
tracer -a:oracle.exe bpf=oracle.exe!_kqvrow,trace:cc
```

```
_kqvrow_        proc near

var_7C          = byte ptr -7Ch
var_18          = dword ptr -18h
var_14          = dword ptr -14h
Dest            = dword ptr -10h
var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4
arg_8           = dword ptr  10h
arg_C           = dword ptr  14h
```

[11] http://yurichev.com/oracle_tables.html

```
arg_14          = dword ptr  1Ch
arg_18          = dword ptr  20h

; FUNCTION CHUNK AT .text1:056C11A0 SIZE 00000049 BYTES

                push    ebp
                mov     ebp, esp
                sub     esp, 7Ch
                mov     eax, [ebp+arg_14] ; [EBP+1Ch]=1
                mov     ecx, TlsIndex   ; [69AEB08h]=0
                mov     edx, large fs:2Ch
                mov     edx, [edx+ecx*4] ; [EDX+ECX*4]=0xc98c938
                cmp     eax, 2          ; EAX=1
                mov     eax, [ebp+arg_8] ; [EBP+10h]=0xcdfe554
                jz      loc_2CE1288
                mov     ecx, [eax]      ; [EAX]=0..5
                mov     [ebp+var_4], edi ; EDI=0xc98c938

loc_2CE10F6:                            ; CODE XREF: _kqvrow_+10A
                                        ; _kqvrow_+1A9
                cmp     ecx, 5          ; ECX=0..5
                ja      loc_56C11C7
                mov     edi, [ebp+arg_18] ; [EBP+20h]=0
                mov     [ebp+var_14], edx ; EDX=0xc98c938
                mov     [ebp+var_8], ebx ; EBX=0
                mov     ebx, eax        ; EAX=0xcdfe554
                mov     [ebp+var_C], esi ; ESI=0xcdfe248

loc_2CE110D:                            ; CODE XREF: _kqvrow_+29E00E6
                mov     edx, ds:off_628B09C[ecx*4] ; [ECX*4+628B09Ch]=0x2ce1116, 0x2ce11ac, 0x2ce11db, 0
    x2ce11f6, 0x2ce1236, 0x2ce127a
                jmp     edx             ; EDX=0x2ce1116, 0x2ce11ac, 0x2ce11db, 0x2ce11f6, 0x2ce1236, 0
    x2ce127a
; ---------------------------------------------------------------------------

loc_2CE1116:                            ; DATA XREF: .rdata:off_628B09C
                push    offset aXKqvvsnBuffer ; "x$kqvvsn buffer"
                mov     ecx, [ebp+arg_C] ; [EBP+14h]=0x8a172b4
                xor     edx, edx
                mov     esi, [ebp+var_14] ; [EBP-14h]=0xc98c938
                push    edx             ; EDX=0
                push    edx             ; EDX=0
                push    50h
                push    ecx             ; ECX=0x8a172b4
                push    dword ptr [esi+10494h] ; [ESI+10494h]=0xc98cd58
                call    _kghalf         ; tracing nested maximum level (1) reached, skipping this CALL
                mov     esi, ds:__imp__vsnnum ; [59771A8h]=0x61bc49e0
                mov     [ebp+Dest], eax ; EAX=0xce2ffb0
                mov     [ebx+8], eax    ; EAX=0xce2ffb0
                mov     [ebx+4], eax    ; EAX=0xce2ffb0
                mov     edi, [esi]      ; [ESI]=0xb200100
                mov     esi, ds:__imp__vsnstr ; [597D6D4h]=0x65852148, "- Production"
                push    esi             ; ESI=0x65852148, "- Production"
                mov     ebx, edi        ; EDI=0xb200100
                shr     ebx, 18h        ; EBX=0xb200100
                mov     ecx, edi        ; EDI=0xb200100
                shr     ecx, 14h        ; ECX=0xb200100
                and     ecx, 0Fh        ; ECX=0xb2
                mov     edx, edi        ; EDI=0xb200100
                shr     edx, 0Ch        ; EDX=0xb200100
                movzx   edx, dl         ; DL=0
                mov     eax, edi        ; EDI=0xb200100
                shr     eax, 8          ; EAX=0xb200100
                and     eax, 0Fh        ; EAX=0xb2001
                and     edi, 0FFh       ; EDI=0xb200100
                push    edi             ; EDI=0
```

```
                mov     edi, [ebp+arg_18] ; [EBP+20h]=0
                push    eax             ; EAX=1
                mov     eax, ds:__imp__vsnban ; [597D6D8h]=0x65852100, "Oracle Database 11g Enterprise
    Edition Release %d.%d.%d.%d.%d %s"
                push    edx             ; EDX=0
                push    ecx             ; ECX=2
                push    ebx             ; EBX=0xb
                mov     ebx, [ebp+arg_8] ; [EBP+10h]=0xcdfe554
                push    eax             ; EAX=0x65852100, "Oracle Database 11g Enterprise Edition Release %d
    .%d.%d.%d.%d %s"
                mov     eax, [ebp+Dest] ; [EBP-10h]=0xce2ffb0
                push    eax             ; EAX=0xce2ffb0
                call    ds:__imp__sprintf ; op1=MSVCR80.dll!sprintf tracing nested maximum level (1) reached
    , skipping this CALL
                add     esp, 38h
                mov     dword ptr [ebx], 1

loc_2CE1192:                            ; CODE XREF: _kqvrow_+FB
                                        ; _kqvrow_+128 ...
                test    edi, edi        ; EDI=0
                jnz     __VInfreq__kqvrow
                mov     esi, [ebp+var_C] ; [EBP-0Ch]=0xcdfe248
                mov     edi, [ebp+var_4] ; [EBP-4]=0xc98c938
                mov     eax, ebx        ; EBX=0xcdfe554
                mov     ebx, [ebp+var_8] ; [EBP-8]=0
                lea     eax, [eax+4]    ; [EAX+4]=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production", "
    Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production", "PL/SQL Release 11.2.0.1.0 -
    Production", "TNS for 32-bit Windows: Version 11.2.0.1.0 - Production"

loc_2CE11A8:                            ; CODE XREF: _kqvrow_+29E00F6
                mov     esp, ebp
                pop     ebp
                retn                    ; EAX=0xcdfe558
; ---------------------------------------------------------------------------

loc_2CE11AC:                            ; DATA XREF: .rdata:0628B0A0
                mov     edx, [ebx+8]    ; [EBX+8]=0xce2ffb0, "Oracle Database 11g Enterprise Edition Release
     11.2.0.1.0 - Production"
                mov     dword ptr [ebx], 2
                mov     [ebx+4], edx    ; EDX=0xce2ffb0, "Oracle Database 11g Enterprise Edition Release
    11.2.0.1.0 - Production"
                push    edx             ; EDX=0xce2ffb0, "Oracle Database 11g Enterprise Edition Release
    11.2.0.1.0 - Production"
                call    _kkxvsn         ; tracing nested maximum level (1) reached, skipping this CALL
                pop     ecx
                mov     edx, [ebx+4]    ; [EBX+4]=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
                movzx   ecx, byte ptr [edx] ; [EDX]=0x50
                test    ecx, ecx        ; ECX=0x50
                jnz     short loc_2CE1192
                mov     edx, [ebp+var_14]
                mov     esi, [ebp+var_C]
                mov     eax, ebx
                mov     ebx, [ebp+var_8]
                mov     ecx, [eax]
                jmp     loc_2CE10F6
; ---------------------------------------------------------------------------

loc_2CE11DB:                            ; DATA XREF: .rdata:0628B0A4
                push    0
                push    50h
                mov     edx, [ebx+8]    ; [EBX+8]=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
                mov     [ebx+4], edx    ; EDX=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
                push    edx             ; EDX=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
                call    _lmxver         ; tracing nested maximum level (1) reached, skipping this CALL
                add     esp, 0Ch
                mov     dword ptr [ebx], 3
```

```
                jmp     short loc_2CE1192
; ---------------------------------------------------------------------------

loc_2CE11F6:                            ; DATA XREF: .rdata:0628B0A8
                mov     edx, [ebx+8]    ; [EBX+8]=0xce2ffb0
                mov     [ebp+var_18], 50h
                mov     [ebx+4], edx    ; EDX=0xce2ffb0
                push    0
                call    _npinli         ; tracing nested maximum level (1) reached, skipping this CALL
                pop     ecx
                test    eax, eax        ; EAX=0
                jnz     loc_56C11DA
                mov     ecx, [ebp+var_14] ; [EBP-14h]=0xc98c938
                lea     edx, [ebp+var_18] ; [EBP-18h]=0x50
                push    edx             ; EDX=0xd76c93c
                push    dword ptr [ebx+8] ; [EBX+8]=0xce2ffb0
                push    dword ptr [ecx+13278h] ; [ECX+13278h]=0xacce190
                call    _nrtnsvrs       ; tracing nested maximum level (1) reached, skipping this CALL
                add     esp, 0Ch

loc_2CE122B:                            ; CODE XREF: _kqvrow_+29E0118
                mov     dword ptr [ebx], 4
                jmp     loc_2CE1192
; ---------------------------------------------------------------------------

loc_2CE1236:                            ; DATA XREF: .rdata:0628B0AC
                lea     edx, [ebp+var_7C] ; [EBP-7Ch]=1
                push    edx             ; EDX=0xd76c8d8
                push    0
                mov     esi, [ebx+8]    ; [EBX+8]=0xce2ffb0, "TNS for 32-bit Windows: Version 11.2.0.1.0 -
    Production"
                mov     [ebx+4], esi    ; ESI=0xce2ffb0, "TNS for 32-bit Windows: Version 11.2.0.1.0 -
    Production"
                mov     ecx, 50h
                mov     [ebp+var_18], ecx ; ECX=0x50
                push    ecx             ; ECX=0x50
                push    esi             ; ESI=0xce2ffb0, "TNS for 32-bit Windows: Version 11.2.0.1.0 -
    Production"
                call    _lxvers         ; tracing nested maximum level (1) reached, skipping this CALL
                add     esp, 10h
                mov     edx, [ebp+var_18] ; [EBP-18h]=0x50
                mov     dword ptr [ebx], 5
                test    edx, edx        ; EDX=0x50
                jnz     loc_2CE1192
                mov     edx, [ebp+var_14]
                mov     esi, [ebp+var_C]
                mov     eax, ebx
                mov     ebx, [ebp+var_8]
                mov     ecx, 5
                jmp     loc_2CE10F6
; ---------------------------------------------------------------------------

loc_2CE127A:                            ; DATA XREF: .rdata:0628B0B0
                mov     edx, [ebp+var_14] ; [EBP-14h]=0xc98c938
                mov     esi, [ebp+var_C] ; [EBP-0Ch]=0xcdfe248
                mov     edi, [ebp+var_4] ; [EBP-4]=0xc98c938
                mov     eax, ebx        ; EBX=0xcdfe554
                mov     ebx, [ebp+var_8] ; [EBP-8]=0

loc_2CE1288:                            ; CODE XREF: _kqvrow_+1F
                mov     eax, [eax+8]    ; [EAX+8]=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production"
                test    eax, eax        ; EAX=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production"
                jz      short loc_2CE12A7
                push    offset aXKqvvsnBuffer ; "x$kqvvsn buffer"
                push    eax             ; EAX=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production"
                mov     eax, [ebp+arg_C] ; [EBP+14h]=0x8a172b4
```

```
                push    eax             ; EAX=0x8a172b4
                push    dword ptr [edx+10494h] ; [EDX+10494h]=0xc98cd58
                call    _kghfrf         ; tracing nested maximum level (1) reached, skipping this CALL
                add     esp, 10h

loc_2CE12A7:                            ; CODE XREF: _kqvrow_+1C1
                xor     eax, eax
                mov     esp, ebp
                pop     ebp
                retn                    ; EAX=0
_kqvrow_        endp
```

Now it's easy to see that row number is passed from outside of function. The function returning the string constructing it as follows:

String 1	Using `vsnstr`, `vsnnum`, `vsnban` global variables. Calling `sprintf()`.
String 2	Calling `kkxvsn()`.
String 3	Calling `lmxver()`.
String 4	Calling `npinli()`, `nrtnsvrs()`.
String 5	Calling `lxvers()`.

That's how corresponding functions are called for determining each module's version.

7.3.2 `X$KSMLRU` table in Oracle RDBMS

There is a mention of some special table in the *Diagnosing and Resolving Error ORA-04031 on the Shared Pool or Other Memory Pools [Video] [ID 146599.1]* note:

> There is a fixed table called X$KSMLRU that tracks allocations in the shared pool that cause other objects in the shared pool to be aged out. This fixed table can be used to identify what is causing the large allocation.
>
> If many objects are being periodically flushed from the shared pool then this will cause response time problems and will likely cause library cache latch contention problems when the objects are reloaded into the shared pool.
>
> One unusual thing about the X$KSMLRU fixed table is that the contents of the fixed table are erased whenever someone selects from the fixed table. This is done since the fixed table stores only the largest allocations that have occurred. The values are reset after being selected so that subsequent large allocations can be noted even if they were not quite as large as others that occurred previously. Because of this resetting, the output of selecting from this table should be carefully kept since it cannot be retrieved back after the query is issued.

However, as it can be easily checked, this table's contents is cleared each time table querying. Are we able to find why? Let's back to tables we already know: `kqftab` and `kqftap` which were generated with oracle tables[12] help, containing all information about X$-tables, now we can see here, the `ksmlrs()` function is called to prepare this table's elements:

Listing 7.10: Result of oracle tables

```
kqftab_element.name: [X$KSMLRU] ?: [ksmlr] 0x4 0x64 0x11 0xc 0xffffc0bb 0x5
kqftap_param.name=[ADDR] ?: 0x917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRIDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRDUR] ?: 0xb02 0x0 0x0 0x0 0x4 0x4 0x0
kqftap_param.name=[KSMLRSHRPOOL] ?: 0xb02 0x0 0x0 0x0 0x4 0x8 0x0
```

[12] http://yurichev.com/oracle_tables.html

```
kqftap_param.name=[KSMLRCOM] ?: 0x501 0x0 0x0 0x0 0x14 0xc 0x0
kqftap_param.name=[KSMLRSIZ] ?: 0x2 0x0 0x0 0x0 0x4 0x20 0x0
kqftap_param.name=[KSMLRNUM] ?: 0x2 0x0 0x0 0x0 0x4 0x24 0x0
kqftap_param.name=[KSMLRHON] ?: 0x501 0x0 0x0 0x0 0x20 0x28 0x0
kqftap_param.name=[KSMLROHV] ?: 0xb02 0x0 0x0 0x0 0x4 0x48 0x0
kqftap_param.name=[KSMLRSES] ?: 0x17 0x0 0x0 0x0 0x4 0x4c 0x0
kqftap_param.name=[KSMLRADU] ?: 0x2 0x0 0x0 0x0 0x4 0x50 0x0
kqftap_param.name=[KSMLRNID] ?: 0x2 0x0 0x0 0x0 0x4 0x54 0x0
kqftap_param.name=[KSMLRNSD] ?: 0x2 0x0 0x0 0x0 0x4 0x58 0x0
kqftap_param.name=[KSMLRNCD] ?: 0x2 0x0 0x0 0x0 0x4 0x5c 0x0
kqftap_param.name=[KSMLRNED] ?: 0x2 0x0 0x0 0x0 0x4 0x60 0x0
kqftap_element.fn1=ksmlrs
kqftap_element.fn2=NULL
```

Indeed, with the *tracer* 5.0.1 help it's easy to see that this function is called each time we query the X$KSMLRU table.

Here we see a references to ksmsplu_sp() and ksmsplu_jp() functions, each of them call ksmsplu() finally. At the end of ksmsplu() function we see a call to memset():

Listing 7.11: ksm.o

```
...

.text:00434C50 loc_434C50:                             ; DATA XREF: .rdata:off_5E50EA8
.text:00434C50                 mov     edx, [ebp-4]
.text:00434C53                 mov     [eax], esi
.text:00434C55                 mov     esi, [edi]
.text:00434C57                 mov     [eax+4], esi
.text:00434C5A                 mov     [edi], eax
.text:00434C5C                 add     edx, 1
.text:00434C5F                 mov     [ebp-4], edx
.text:00434C62                 jnz     loc_434B7D
.text:00434C68                 mov     ecx, [ebp+14h]
.text:00434C6B                 mov     ebx, [ebp-10h]
.text:00434C6E                 mov     esi, [ebp-0Ch]
.text:00434C71                 mov     edi, [ebp-8]
.text:00434C74                 lea     eax, [ecx+8Ch]
.text:00434C7A                 push    370h            ; Size
.text:00434C7F                 push    0               ; Val
.text:00434C81                 push    eax             ; Dst
.text:00434C82                 call    __intel_fast_memset
.text:00434C87                 add     esp, 0Ch
.text:00434C8A                 mov     esp, ebp
.text:00434C8C                 pop     ebp
.text:00434C8D                 retn
.text:00434C8D _ksmsplu        endp
```

Constructions like memset (block, 0, size) are often used just to zero memory block. What if we would take a risk, block memset() call and see what will happen?

Let's run *tracer* 5.0.1 with the following options: set breakpoint at 0x434C7A (the point where memset() arguments are to be passed), thus, that *tracer* 5.0.1 set program counter EIP at this place to the place where passed to memset() arguments are to be cleared (at 0x434C8A) It can be said, we just simulate an unconditional jump from the address 0x434C7A to 0x434C8A.

```
tracer -a:oracle.exe bpx=oracle.exe!0x00434C7A,set(eip,0x00434C8A)
```

(Important: all these addresses are valid only for win32-version of Oracle RDBMS 11.2)

Indeed, now we can query X$KSMLRU table as many times as we want and it's not cleared anymore!

~~Don't try this at home ("MythBusters")~~ Don't try this on your production servers.

It's probably not a very useful or desired system behaviour, but as an experiment of locating piece of code we need, that's perfectly suit our needs!

7.3.3 V$TIMER table in Oracle RDBMS

V$TIMER is another *fixed view*, reflecting some often changed value:

> V$TIMER displays the elapsed time in hundredths of a second. Time is measured since the beginning of the epoch, which is operating system specific, and wraps around to 0 again whenever the value overflows four bytes (roughly 497 days).

(From Oracle RDBMS documentation [13])

It's interesting that periods are different for Oracle for win32 and for Linux. Will we able to find a function generating this value?

As we can see, this information is finally taken from X$KSUTM table.

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='V$TIMER';

VIEW_NAME
------------------------------
VIEW_DEFINITION
--------------------------------------------------------------------------------

V$TIMER
select  HSECS from GV$TIMER where inst_id = USERENV('Instance')

SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='GV$TIMER';

VIEW_NAME
------------------------------
VIEW_DEFINITION
--------------------------------------------------------------------------------

GV$TIMER
select inst_id,ksutmtim from x$ksutm
```

Now we stuck in problem, there are no references to value generating function(s) in the tables kqftab/kqftap:

Listing 7.12: Result of oracle tables

```
kqftab_element.name: [X$KSUTM] ?: [ksutm] 0x1 0x4 0x4 0x0 0xffffc09b 0x3
kqftap_param.name=[ADDR] ?: 0x10917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0x20b02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSUTMTIM] ?: 0x1302 0x0 0x0 0x0 0x4 0x0 0x1e
kqftap_element.fn1=NULL
kqftap_element.fn2=NULL
```

Let's try to find a string KSUTMTIM, and we find it in this function:

```
kqfd_DRN_ksutm_c proc near              ; DATA XREF: .rodata:0805B4E8

arg_0           = dword ptr  8
arg_8           = dword ptr  10h
arg_C           = dword ptr  14h

                push    ebp
                mov     ebp, esp
                push    [ebp+arg_C]
                push    offset ksugtm
                push    offset _2__STRING_1263_0 ; "KSUTMTIM"
                push    [ebp+arg_8]
                push    [ebp+arg_0]
```

[13] http://docs.oracle.com/cd/B28359_01/server.111/b28320/dynviews_3104.htm

```
                call    kqfd_cfui_drain
                add     esp, 14h
                mov     esp, ebp
                pop     ebp
                retn
kqfd_DRN_ksutm_c endp
```

The function `kqfd_DRN_ksutm_c()` is mentioned in `kqfd_tab_registry_0` table:

```
dd offset _2__STRING_62_0 ; "X$KSUTM"
dd offset kqfd_OPN_ksutm_c
dd offset kqfd_tabl_fetch
dd 0
dd 0
dd offset kqfd_DRN_ksutm_c
```

There are also some function `ksugtm()` referenced here. Let's see what's in it (Linux x86):

Listing 7.13: ksu.o

```
ksugtm          proc near

var_1C          = byte ptr -1Ch
arg_4           = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
                sub     esp, 1Ch
                lea     eax, [ebp+var_1C]
                push    eax
                call    slgcs
                pop     ecx
                mov     edx, [ebp+arg_4]
                mov     [edx], eax
                mov     eax, 4
                mov     esp, ebp
                pop     ebp
                retn
ksugtm          endp
```

Almost the same code in win32-version.
Is this the function we are looking for? Let's see:

```
tracer -a:oracle.exe bpf=oracle.exe!_ksugtm,args:2,dump_args:0x4
```

Let's try again:

```
SQL> select * from V$TIMER;

     HSECS
----------
  27294929

SQL> select * from V$TIMER;

     HSECS
----------
  27295006

SQL> select * from V$TIMER;

     HSECS
----------
  27295167
```

Output of *tracer* 5.0.1:

```
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!__VInfreq__qerfxFetch+0xfad (0
    x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                                           "8.              "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: D1 7C A0 01                                     ".|..            "
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!__VInfreq__qerfxFetch+0xfad (0
    x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                                           "8.              "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: 1E 7D A0 01                                     ".}..            "
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!__VInfreq__qerfxFetch+0xfad (0
    x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                                           "8.              "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: BF 7D A0 01                                     ".}..            "
```

Indeed — the value is the same we see in SQL*Plus and it's returning via second argument.
Let's see what is in `slgcs()` (Linux x86):

```
slgcs           proc near

var_4           = dword ptr -4
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                push    esi
                mov     [ebp+var_4], ebx
                mov     eax, [ebp+arg_0]
                call    $+5
                pop     ebx
                nop                     ; PIC mode
                mov     ebx, offset _GLOBAL_OFFSET_TABLE_
                mov     dword ptr [eax], 0
                call    sltrgatime64    ; PIC mode
                push    0
                push    0Ah
                push    edx
                push    eax
                call    __udivdi3       ; PIC mode
                mov     ebx, [ebp+var_4]
                add     esp, 10h
                mov     esp, ebp
                pop     ebp
                retn
slgcs           endp
```

(it's just a call to `sltrgatime64()` and division of its result by 10 1.12)
And win32-version:

```
_slgcs          proc near               ; CODE XREF: _dbgefgHtElResetCount+15
                                        ; _dbgerRunActions+1528
                db      66h
                nop
                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+8]
                mov     dword ptr [eax], 0
                call    ds:__imp__GetTickCount@0 ; GetTickCount()
                mov     edx, eax
```

```
                mov     eax, 0CCCCCCCDh
                mul     edx
                shr     edx, 3
                mov     eax, edx
                mov     esp, ebp
                pop     ebp
                retn
_slgcs          endp
```

It's just result of `GetTickCount()` [14] divided by 10 1.12.

Voilà! That's why win32-version and Linux x86 version show different results, just because they are generated by different operation system functions.

Drain apparently mean *connecting* specific table column to specific function.

I added the table `kqfd_tab_registry_0` to oracle tables[15], now we can see, how table column's variables are *connected* to specific functions:

```
[X$KSUTM] [kqfd_OPN_ksutm_c] [kqfd_tabl_fetch] [NULL] [NULL] [kqfd_DRN_ksutm_c]
[X$KSUSGIF] [kqfd_OPN_ksusg_c] [kqfd_tabl_fetch] [NULL] [NULL] [kqfd_DRN_ksusg_c]
```

OPN, apparently, *open*, and *DRN*, apparently, meaning *drain*.

[14] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724408(v=vs.85).aspx
[15] http://yurichev.com/oracle_tables.html

Chapter 8

Other things

8.1 Compiler's anomalies

Intel C++ 10.1, which was used for Oracle RDBMS 11.2 Linux86 compilation, may emit two JZ in row, and there are no references to the second JZ. Second JZ is thus senseless.

Listing 8.1: kdli.o from 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
```

Listing 8.2: from the same code

```
.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
```

It's probably code generator bug wasn't found by tests, because, resulting code is working correctly anyway.

Another compiler anomaly I described here 1.15.2.

I'm showing such cases here, so to understand that such compilers errors are possible and sometimes one should not to rack one's brain and think why compiler generated such strange code.

Chapter 9

Tasks solutions

9.1 Easy level

9.1.1 Task 1.1

Solution: `toupper()`.

C source code:

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

9.1.2 Task 1.2

Solution: `atoi()`

C source code:

```
#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 Task 1.3

Solution: `srand()` / `rand()`.

C source code:

```
static unsigned int v;

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

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

9.1.4 Task 1.4

Solution: `strstr()`.

C source code:

```
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 Task 1.5

Hint #1: Keep in mind that `__v` — global variable.

Hint #2: That function is called in startup code, before `main()` execution.

Solution: early Pentium CPU FDIV bug checking[1].

C source code:

```
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 Task 1.6

Hint: it might be helpful to google a constant used here.

Solution: TEA encryption algorithm[2].

C source code (taken from 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 Task 1.7

Hint: the table contain pre-calculated values. It's possible to implement the function without it, but it will work slower, though.

Solution: this function reverse all bits in input 32-bit integer. It's `lib/bitrev.c` from Linux kernel.

C source code:

```
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 Task 1.8

Solution: two 100*200 matrices of *double* type addition.

C/C++ source code:

```
#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 Task 1.9

Solution: two matrices (one is 100*200, second is 100*300) of *double* type multiplication, result: 100*300 matrix.

C/C++ source code:

```
#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 Middle level

9.2.1 Task 2.1

Hint #1: The code has one characteristic thing, considering it, it may help narrowing search of right function among glibc functions.

Solution: characteristic — is callback-function calling 1.19, pointer to which is passed in 4th argument. It's `quicksort()`.

C source code:

```
/* 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;
                }
            }
        }
    }
}
```

Afterword

9.3 Donate

This book is free, available freely and available in source code form[3] (LaTeX), and it will be so forever.

If you want to support my work, so that I could continue to add things to it regularly, you may consider donation.

You may donate by sending small (or not small) donation to bitcoin[4]
`1HRGTRdFNH1cE81zxWQg6jTtkLzAiGU9Lp`

Figure 9.1: bitcoin address

Other ways to donate are available on the page: http://yurichev.com/donate.html
Major benefactors will be mentioned right here.

9.4 Questions?

Do not hesitate to mail any questions to the author: <dennis@yurichev.com>

Please, also do not hesitate to send me any corrections (including grammar ones (you see how horrible my English is?)), etc.

[3] https://github.com/dennis714/RE-for-beginners
[4] http://en.wikipedia.org/wiki/Bitcoin

Bibliography

[App10] Apple. iOS ABI Function Call Guide. 2010. Also available as http://developer.apple.com/library/ios/documentation/Xcode/Conceptual/iPhoneOSABIReference/iPhoneOSABIReference.pdf.

[Cli] Marshall Cline. C++ faq. Also available as http://www.parashift.com/c++-faq-lite/index.html.

[ISO07] ISO. ISO/IEC 9899:TC3 (C C99 standard). 2007. Also available as http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf.

[ISO13] ISO. ISO/IEC 14882:2011 (C++ 11 standard). 2013. Also available as http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf.

[Ker88] Brian W. Kernighan. The C Programming Language. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[Knu98] Donald E. Knuth. The Art of Computer Programming Volumes 1-3 Boxed Set. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1998.

[Loh10] Eugene Loh. The ideal hpc programming language. Queue, 8(6):30:30–30:38, June 2010.

[Ltd94] Advanced RISC Machines Ltd. The ARM Cookbook. 1994. Also available as http://yurichev.com/ref/ARM%20Cookbook%20(1994).

[Ray03] Eric S. Raymond. The Art of UNIX Programming. Pearson Education, 2003. Also available as http://catb.org/esr/writings/taoup/html/.

[Rit86] Dennis M. Ritchie. Where did ++ come from? (net.lang.c). http://yurichev.com/mirrors/c_dmr_postincrement.txt, 1986. [Online; accessed 2013].

[Rit93] Dennis M. Ritchie. The development of the c language. SIGPLAN Not., 28(3):201–208, March 1993. Also available as http://yurichev.com/mirrors/dmr-The%20Development%20of%20the%20C%20Language-1993.pdf.

[War02] Henry S. Warren. Hacker's Delight. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

Index