Quick introduction to reverse engineering for beginners

Dennis Yurichev <dennis@yurichev.com>

March 4, 2018

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 code created by C/C++ compilers (which is a most large mass of all executable software in the world).

There are two most used compilers: MSVC and GCC, these we will use for experiments.

There are two most used x86 assembler syntax: Intel (most used in DOS/Windows) and AT&T (used in *NIX) [1]. Here we use Intel syntax. IDA 6 produce Intel syntax listings too.

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

Chapter 2

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 assembler, 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 other, so I'll try to describe here some examples.

2.1 Hello, world!

Let's start with famous example from the book "The C programming Language"[1]:

```
#include <stdio.h>

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

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

```
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 this string somehow, so it define internal name `$SG3830` for it.

As we can see, the string is terminated by zero byte — this is C/C++ standard of 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 3.2.

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.

Since it is 32-bit code, we need exactly 4 bytes for address passing through the stack. It's 8 bytes in x64-code

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

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

More about stack in relevant section 2.2.

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`

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

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.

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 6 disassembler assistance, let's see how main() function was created.

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

```
main            proc near

var_10          = dword ptr -10h

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

Almost the same, only sole exception that in function prologue we see AND ESP, 0FFFFFFF0h — this instruction aligning ESP value on 16-byte border, resulting some values in stack aligned too.

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

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.

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

2.2 Stack

Stack — is one of the most fundamental things in computer science.[4].

Technically, this is just piece of process memory + ESP register as a pointer within piece of memory.

Most frequently used stack access instructions are PUSH and POP. PUSH subtracting ESP by 4 and then writing contents of its sole operand to the memory address pointing by ESP.

POP is reverse operation: get a data from memory pointing by ESP and then add 4 to ESP. Of course, this is for 32-bit environment. 8 will be here instead of 4 in x64 environment.

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

What stack is used for?

2.2.1 Save return address where function should return control after execution

While calling another function by CALL instruction, the address of point exactly after CALL 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
; 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*[5]:

```
?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.

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

[5] irony here

2.2.2 Function arguments passing

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

Callee[6] function get its arguments via `ESP` ponter.

See also section about calling conventions 3.5.

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

However, it is convenient tradition to use stack for this.

2.2.3 Local variable storage

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

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

alloca() function

It is worth noting `alloca()` function.[7].

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 3.2 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:

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

    puts (buf);
};
```

Let's compile (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
```

[6]Function being called

[7]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`

```
    call    _puts
    add     esp, 28         ; 0000001cH

...
```

The sole `alloca()` argument passed via `EAX` (but not via stack).

After `alloca()` call, `ESP` is not 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:

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

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

                push    ebp
                mov     ebp, esp
                sub     esp, 38h
                mov     eax, large gs:14h
                mov     [ebp+var_C], eax
                xor     eax, eax
                sub     esp, 624
                lea     eax, [esp+18h]
                add     eax, 0Fh
                shr     eax, 4           ; 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
```

2.2.4 (Windows) SEH

SEH (*Structured Exception Handling*) records are also stored in stack (if needed). [8].

2.2.5 Buffer overflow protection

More about it here 2.13.1.

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

2.3 printf() with several arguments

Now let's extend *Hello, world!* 2.1 example, replacing `printf()` in `main()` function body by this:

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

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.

Here 4 arguments. 4*4 = 16 — exactly 16 byte they occupy in stack: 32-bit pointer to string and 3 number of *int* type.

Variables of *int* type in 32-bit environment has 32-bit width.

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 3.5.

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 6 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`.

By the way, this difference 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.

2.4 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*.

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 handy.

Function scanf() in our example has two arguments.

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

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(). Indeed: printf() with only 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.

Зачем GCC заменил один вызов на другой? Потому что puts() работает быстрее [9].

It working faster because just passes characters to stdout not comparing each with % symbol.

Why scanf() is renamed to ___isoc99_scanf, I do not know yet.

As before — arguments are placed into stack by MOV instruction.

2.4.1 Global variables

What if x variable from previous example will not 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
```

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

```
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 typed as DWORD (DD meaning DWORD = 32 bit).

If you will open compiled .exe in IDA 6, 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 6 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[10] file format this segment has such attributes:

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

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

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

2.4.2 scanf() result checking

As I said before, it is not very clever 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()`[11] 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.

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

What we got in assembler (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()`) must have access to result of callee function (`scanf()`), so callee leave this value in `EAX` register.

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

[11] MSDN: scanf, wscanf

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`.

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[12]. `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 result zero, `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.

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

2.5 Passing arguments via stack

Now we figured out that caller function passing arguments to callee via stack. But how callee[13] 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;
};
```

What we have after compilation (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 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[14] of `EAX` and what

[13] function being called

[14] result of multiplication

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:

```
                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.

2.6 One more word about results returning.

Function execution result is often returned[15] in EAX register. If it's byte type — then in lowest register EAX part — AL. If function returning *float* number, FPU register ST(0) will be used instead.

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.

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

2.7 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;
};
```

What we have in `f_signed()` function:

```
_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 at all. But, without special intervention, it is nearly impossible.

GCC 4.4.1 produce almost the same code, but with puts() 2.4 instead of printf().

Now let's take a look of f_unsigned() produced by 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) is different from JG/JGE/JL/JLE in that way, it works with unsigned numbers.

See also section about signed number representations 3.4. 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:

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

2.8 switch()/case/default

2.8.1 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;
    };
};
```

Result (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
```

Nothing specially new to us, 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`

```
_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()`[16]. And of course, it's all done for the sake of speed.

2.8.2 A lot of cases

If `switch()` statement contain a lot of case's, it is not very handy 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;
```

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

```
    case 4: printf ("four\n"); break;
    default: printf ("something unknown\n"); break;
    };
};
```

We got (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 generated by compiler. 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.

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 3.3 is assembler 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 value from memory through bus, through cache memory, etc, in much effective way if it is aligned.

Let's see what GCC 4.4.1 generate:

```
                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) 2.14.3. Then label address is taken from `off_804855C` array, address calculated and stored into `EAX`, then `JMP EAX` do actual jump.

2.9 Loops

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 and pass control flow to label mentioned in `LOOP` operand. Probably, this instruction is not very handy, 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 hand-written piece of assembler code.

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

In C/C++ loops are defined by `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):

```
_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:

```
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`):

```
_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 much local variables are used.

One very important property is that `f()` function should not change `ESI` value, in fact, it should not use that register at all. 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):

```
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 that 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:

```
                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()`.

2.10 strlen()

Now let's talk about loops one more time. Often, `strlen()` function[17] is implemented using `while()` statement. Here is how it done in MSVC standard libraries:

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

        while( *eos++ ) ;

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

Let's compile:

```
п»ї_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` 2.10 and `TEST`.

About first: `MOVSX` 2.10 is intended to take byte from some place and store value in 32-bit register. `MOVSX` 2.10 meaning *MOV with Sign-Extent*. Rest bits starting at 8th till 31th `MOVSX` 2.10 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` 2.10 does.

See also more in section *Signed number representations* 3.4.

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

[17] counting characters in string in C language

Then we see `TEST EDX, EDX`. About `TEST` instruction, read more in section about bit fields 2.14. 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` 2.10. `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 handy only because it is able 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
```

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 3.4.

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.

2.11 Division by 9

Very simple function:

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

Is compiled in a very predictable way:

```
_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[18], 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 2.10 does. If we turn optimization on (/Ox), we got:

```
_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 [19] 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:

```
        public f
f       proc near

arg_0   = dword ptr  8

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

[18] result of division

[19] More about division by multiplication: MSDN: Integer division by constants, http://www.nynaeve.net/?p=115

```
        pop     ebp
        retn
f       endp
```

2.12 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[20]before FPU studying, or learn Forth language basics[21].

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.

From the 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[22]format.

C/C++ language offer at least two floating number types, *float* (*single-precision*[23], 32 bits) [24] and *double* (*double-precision*[25], 64 bits).

GCC also supports *long double* type (*extended precision*[26], 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.

2.12.1 Simple example

Let's consider simple example:

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

Compile it in MSVC 2010:

```
¤»ïCONST    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

    fld    QWORD PTR _b$[ebp]

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

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

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

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

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

[24] single precision float numbers format is also addressed in *Working with the float type as with a structure* 2.15.6 section

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

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

```
    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[27].

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[28], you will quickly understand that this is stack machine[29].

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:

```
¤»ï                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]

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

                pop     ebp
                faddp   st(1), st
```

[27] division result

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

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

```
; 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 piece of disassembled code was produced using IDA 6 which named `ST(0)` register as `ST` for short.

2.12.2 Passing floating point number via arguments

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

        return 0;
}
```

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

```
п»їCONST    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()`[30] 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.

2.12.3 Comparison example

Let's try this:

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

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

        return b;
};
```

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

MSVC 2010 generated:

```
¤»ïPUBLIC    _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 [31] have not 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 differentiate if 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.

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

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

Now let's talk about parity flag. Yet 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.[32]

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

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 remembered 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`

```
п»ї_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 different 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` и `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

```
п»їd_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[33]. 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).

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 remember, 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 что `SETcc`[34] 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

```
п»ї                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)
```

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

[34] *cc* is *condition code*

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

                fucom   st(1) ; compare _a and _b
                fnstsw  ax
                sahf
                ja      short loc_8048448
                fstp    st ; store ST(0) to ST(0) (idle operation), pop value at top of
                    stack, leave _b at top
                jmp     short loc_804844A

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

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.

2.13 Arrays

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

```
#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;
};
```

Let's compile:

```
_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 2.14.3.

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

Here is what GCC 4.4.1 does:

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

var_70          = dword ptr -70h
var_6C          = dword ptr -6Ch
var_68          = dword ptr -68h
i_2             = dword ptr -54h
i               = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 70h
                mov     [esp+70h+i], 0          ; i=0
                jmp     short loc_804840A

loc_80483F7:
                mov     eax, [esp+70h+i]
                mov     edx, [esp+70h+i]
                add     edx, edx                ; edx=i*2
                mov     [esp+eax*4+70h+i_2], edx
                add     [esp+70h+i], 1          ; i++

loc_804840A:
                cmp     [esp+70h+i], 13h
                jle     short loc_80483F7
                mov     [esp+70h+i], 0
                jmp     short loc_8048441

loc_804841B:
                mov     eax, [esp+70h+i]
                mov     edx, [esp+eax*4+70h+i_2]
                mov     eax, offset aADD ; "a[%d]=%d\n"
                mov     [esp+70h+var_68], edx
                mov     edx, [esp+70h+i]
                mov     [esp+70h+var_6C], edx
                mov     [esp+70h+var_70], eax
                call    _printf
                add     [esp+70h+i], 1

loc_8048441:
                cmp     [esp+70h+i], 13h
                jle     short loc_804841B
                mov     eax, 0
                leave
                retn
main            endp
```

2.13.1 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*, 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[35], 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];
```

[35] Java, Python, etc

```
        int i;

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

        return 0;
};
```

And what we've got:

```
п»ї_TEXT    SEGMENT
_i$ = -84                ; size = 4
_a$ = -80                ; size = 80
_main       PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 84                     ; 00000054H
    mov     DWORD PTR _i$[ebp], 0
    jmp     SHORT $LN3@main
$LN2@main:
    mov     eax, DWORD PTR _i$[ebp]
    add     eax, 1
    mov     DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp     DWORD PTR _i$[ebp], 30      ; 0000001eH
    jge     SHORT $LN1@main
    mov     ecx, DWORD PTR _i$[ebp]
    mov     edx, DWORD PTR _i$[ebp]        ; that insruction is obviously redundant
    mov     DWORD PTR _a$[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* 6.0.1, 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
```

So, please keep an eye 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 (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[36]-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*[37].

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 started[38].

There are several methods to protect against it, regardless of C/C++ programmers' negligence. MSVC has options like[39]:

```
/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.

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
```

[36] C Run-Time

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

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

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

```
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.

2.13.2 One more word about arrays

Now we understand, why it's not possible to write something like that in C/C++ code [40]:

```
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.

2.13.3 Multidimensional arrays

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

Let's see:

```
#include <stdio.h>

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

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

We got (MSVC 2010):

```
_DATA    SEGMENT
COMM     _a:DWORD:01770H
_DATA    ENDS
PUBLIC   _insert
_TEXT    SEGMENT
_x$ = 8                 ; size = 4
_y$ = 12                ; size = 4
```

[40] GCC can actually do this by allocating array dynammically in stack (like alloca()), but it's not standard langauge extension

```
_z$ = 16                ; size = 4
_value$ = 20            ; size = 4
_insert    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _x$[ebp]
    imul   eax, 2400                ; 00000960H
    mov    ecx, DWORD PTR _y$[ebp]
    imul   ecx, 120                 ; 00000078H
    lea    edx, DWORD PTR _a[eax+ecx]
    mov    eax, DWORD PTR _z$[ebp]
    mov    ecx, DWORD PTR _value$[ebp]
    mov    DWORD PTR [edx+eax*4], ecx
    pop    ebp
    ret    0
_insert    ENDP
_TEXT      ENDS
```

Nothing special. For index calculation, three input arguments are multiplying in such way to represent array as multidimensional.

GCC 4.4.1:

```
                public insert
insert          proc near

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

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

2.14 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.

2.14.1 Specific bit checking

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):

```
        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:

```
#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()`[41] function.

How `CreateFile()` will check flags?

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

```
.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 2.4.2).

This piece 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 is possible 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:

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

void main()
{
        int handle;
```

[41] MSDN: CreateFile function

```
        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
```

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

```
.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 are probably 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 3.5.3. This works faster, because CPU not needed to access a stack in memory to read argument values. GCC has option *regparm*[42], 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 [43] [44].

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 6, find the `do_filp_open()` function. At the beginning, we will see (comments are mine):

```
do_filp_open    proc near
...
                push    ebp
                mov     ebp, esp
                push    edi
                push    esi
                push    ebx
                mov     ebx, ecx
                add     ebx, 1
                sub     esp, 98h
                mov     esi, [ebp+arg_4] ; acc_mode (5th arg)
```

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

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

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

```
                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
```

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 piece 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
```

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.

2.14.2 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;
};
```

We got (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:

```
_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:

```
                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:

```
                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.

```
                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 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*[45].

2.14.3 Shifts

Bit shifts in C/C++ are implemented via « and » 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«i statement will be counting from 1 to 0x80000000. Describing this operation in naturaul language, we would say *shift 1 by n bits left*. In other words, 1«i statement will consequentially 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.

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 123, that's why it always working correctly.

Let's compile (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
```

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

```
    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:

```
                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):

```
_a$ = 8                                                 ; size = 4
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        shr     eax, 2
        ret     0
_f      ENDP
```

SHR (*SHift Right*) instruction 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.

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 will be division remainder). 2 is leaving after operation as a quotient [46].

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).

2.14.4 CRC32 calculation example

This is very popular table-based CRC32 hash calculation method[47].

```
/* By Bob Jenkins, (c) 2006, Public Domain */

#include <stdio.h>
#include <stddef.h>
#include <string.h>

typedef  unsigned long  ub4;
typedef  unsigned char  ub1;

static const ub4 crctab[256] = {
  0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419, 0x706af48f,
  0xe963a535, 0x9e6495a3, 0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988,
  0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91, 0x1db71064, 0x6ab020f2,
  0xf3b97148, 0x84be41de, 0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7,
  0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0x63066cd9,
  0xfa0f3d63, 0x8d080df5, 0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172,
  0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b, 0x35b5a8fa, 0x42b2986c,
  0xdbbbc9d6, 0xacbcf940, 0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59,
  0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423,
  0xcfba9599, 0xb8bda50f, 0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
  0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d, 0x76dc4190, 0x01db7106,
  0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
  0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818, 0x7f6a0dbb, 0x086d3d2d,
  0x91646c97, 0xe6635c01, 0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e,
  0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0x12b7e950,
  0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
  0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2, 0x4adfa541, 0x3dd895d7,
  0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0,
  0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9, 0x5005713c, 0x270241aa,
  0xbe0b1010, 0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
  0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17, 0x2eb40d81,
  0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a,
  0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683, 0xe3630b12, 0x94643b84,
  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,
```

[46] division result

[47] Source code was taken here: http://burtleburtle.net/bob/c/crc.c

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

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 3.3.

2.15 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.

2.15.1 SYSTEMTIME example

Let's take SYSTEMTIME[48]win32 structure describing time.

That's how it's defined:

```
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):

```
_t$ = -16           ; size = 16
_main       PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 16     ; 00000010H
    lea     eax, DWORD PTR _t$[ebp]
    push    eax
    call    DWORD PTR __imp__GetSystemTime@4
    movzx   ecx, WORD PTR _t$[ebp+12] ; wSecond
    push    ecx
    movzx   edx, WORD PTR _t$[ebp+10] ; wMinute
    push    edx
    movzx   eax, WORD PTR _t$[ebp+8] ; wHour
    push    eax
    movzx   ecx, WORD PTR _t$[ebp+6] ; wDay
    push    ecx
    movzx   edx, WORD PTR _t$[ebp+2] ; wMonth
    push    edx
    movzx   eax, WORD PTR _t$[ebp] ; wYear
    push    eax
    push    OFFSET $SG78811 ; '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
    call    _printf
    add     esp, 28       ; 0000001cH
    xor     eax, eax
```

[48] MSDN: SYSTEMTIME structure

```
    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).

Take a note: the structure beginning with `wYear` field. It can be said, an pointer to SYSTEMTIME structure is passed to `GetSystemTime()`[49], 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.

2.15.2 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.

```
_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
```

[49] MSDN: SYSTEMTIME structure

```
    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` 2.10, 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.

2.15.3 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 6 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` 3.1.

2.15.4 Fields packing in structure

One important thing is fields packing in structures[50].

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).

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
```

[50] See also: Data structure alignment

```
    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*).

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

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[51] and GCC[52].

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:

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

And this:

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

The file PshPack1.h looks like:

```
#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)) */
```

[51] MSDN: Working with Packing Structures

[52] Structure-Packing Pragmas

That's how compiler will pack structures defined after `#pragma pack`.

2.15.5 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):

```
_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 assembler 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.

2.15.6 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`[53]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:

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[54].

```
#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);
```

[53] http://en.wikipedia.org/wiki/CPUID
[54] More about internal GCC assembler

```
    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.

Let's compile it in MSVC 2008 with /Ox option:

```
_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.

```
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 2.12, 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*.

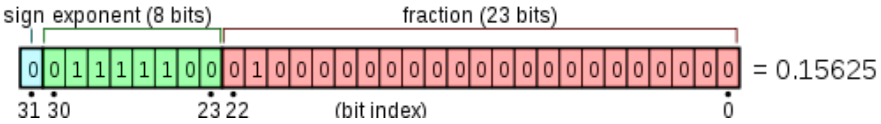

Figure 2.1: float value format (illustration taken from wikipedia)

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

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

float f(float _in)
{
    float f=_in;
    struct float_as_struct t;

    assert (sizeof (struct float_as_struct) == sizeof (float));

    memcpy (&t, &f, sizeof (float));

    t.sign=1; // set negative sign
    t.exponent=t.exponent+2; // multiple d by 2^n (n here is 2)

    memcpy (&f, &t, sizeof (float));

    return f;
};

int main()
{
    printf ("%f\n", f(1.234));
};
```

`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:

```
_t$ = -8            ; size = 4
_f$ = -4            ; size = 4
__in$ = 8           ; size = 4
?f@@YAMM@Z PROC     ; f
    push   ebp
    mov    ebp, esp
    sub    esp, 8

    fld    DWORD PTR __in$[ebp]
    fstp   DWORD PTR _f$[ebp]

    push   4
    lea    eax, DWORD PTR _f$[ebp]
    push   eax
    lea    ecx, DWORD PTR _t$[ebp]
    push   ecx
    call   _memcpy
    add    esp, 12          ; 0000000cH

    mov    edx, DWORD PTR _t$[ebp]
    or     edx, -2147483648 ; 80000000H - set minus sign
    mov    DWORD PTR _t$[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
        exponent
    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?

```
; f(float)
                public _Z1ff
_Z1ff           proc near

var_4           = dword ptr -4
arg_0           = dword ptr  8

                push    ebp
```

```
                mov     ebp, esp
                sub     esp, 4
                mov     eax, [ebp+arg_0]
                or      eax, 80000000h  ; set minus sign
                mov     edx, eax
                and     eax, 807FFFFFh  ; leave only significand and exponent in EAX
                shr     edx, 23         ; prepare exponent
                add     edx, 2          ; add 2
                movzx   edx, dl         ; clear all bits except 7:0 in EAX
                shl     edx, 23         ; shift new calculated exponent to its place
                or      eax, edx        ; add newe exponent and original value without
                    exponent
                mov     [ebp+var_4], eax
                fld     [ebp+var_4]
                leave
                retn
_Z1ff           endp

                public main
main            proc near               ; DATA XREF: _start+17
                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                fld     ds:dword_8048614 ; -4.936
                fstp    qword ptr [esp+8]
                mov     dword ptr [esp+4], offset asc_8048610 ; "%f\n"
                mov     dword ptr [esp], 1
                call    ___printf_chk
                xor     eax, eax
                leave
                retn
main            endp
```

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!

2.16 C++ classes

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, couple of constructors and one method:

```
#include <stdio.h>

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

    c(int a, int b) // ctor
    {
        v1=a;
        v2=b;
    };

    void dump()
    {
        printf ("%d; %d\n", v1, v2);
    };
};

int main()
{
    class c c1;
    class c c2(5,6);

    c1.dump();
    c2.dump();

    return 0;
};
```

Here is how `main()` function looks like translated into assembler:

```
_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 3.5.4 — 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, for example, via first function argument (like GCC).

Why these functions has so odd names? That's *name mangling*[55].

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 allows 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:

```
_this$ = -4              ; size = 4
?dump@c@@QAEXXZ PROC     ; c::dump, COMDAT
```

[55] Wikipedia: Name mangling

```
; _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 after second constructor calling wasn't corrected with `add esp, X`. Please also note that, constructor has `ret 8` instead of `RET` at the end.

That's all because here used thiscall 3.5.4 calling convention, the method of passing values through the stack, which is, together with stdcall 3.5.2 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 3.5.

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.

2.16.1 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[56] 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 sole) argument.

Second constructor:

```
                public _ZN1cC1Eii
_ZN1cC1Eii      proc near

arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch
arg_8           = dword ptr  10h
```

[56] One more document about different compilers name mangling types: http://www.agner.org/optimize/calling_conventions.pdf

```
                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).

2.17 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 [57].

Well-known examples are:

- `qsort()`[58], `atexit()`[59] from the standard C library;
- signals in *NIX OS[60];
- thread starting: `CreateThread()` (win32), `pthread_create()` (POSIX);
- a lot of win32 functions, for example `EnumChildWindows()`[61].

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:

```
__a$ = 8               ; size = 4
__b$ = 12              ; size = 4
_comp       PROC
```

[57] http://en.wikipedia.org/wiki/Callback_(computer_science)
[58] http://en.wikipedia.org/wiki/Qsort_(C_standard_library)
[59] http://www.opengroup.org/onlinepubs/009695399/functions/atexit.html
[60] http://en.wikipedia.org/wiki/Signal.h
[61] 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):

```
.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.

2.17.1 GCC

Not a big difference:

```
                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: (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]
...
```

2.18 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[62] — 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/

2.18.1 Vectorization

Vectorization[63], 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.

/IFRUНапример:For example:

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

This piece 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* (*Перемножить запакованные знаковые DWORD и сохранить младшую часть результата*)and `PMULHW` (*Перемножить апакованные знаковые DWORD и сохранить старшую часть результата*), it's possible to get 4 64-bit products[64] at once.

Thus, loop body count is 1024/4 instead of 1024, that's 4 times less and, of course, faster.

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

[63] Wikipedia: vectorization

[64] multiplication result

Some compilers can do vectorization automatically in some simple cases, for example, Intel C++[65]. I wrote tiny function:

```
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 6):

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

var_10          = dword ptr -10h
sz              = dword ptr  4
ar1             = dword ptr  8
ar2             = dword ptr  0Ch
ar3             = dword ptr  10h

                push    edi
                push    esi
                push    ebx
                push    esi
                mov     edx, [esp+10h+sz]
                test    edx, edx
                jle     loc_15B
                mov     eax, [esp+10h+ar3]
                cmp     edx, 6
                jle     loc_143
                cmp     eax, [esp+10h+ar2]
                jbe     short loc_36
                mov     esi, [esp+10h+ar2]
                sub     esi, eax
                lea     ecx, ds:0[edx*4]
                neg     esi
                cmp     ecx, esi
                jbe     short loc_55

loc_36:                                 ; CODE XREF: f(int,int *,int *,int *)+21
                cmp     eax, [esp+10h+ar2]
                jnb     loc_143
                mov     esi, [esp+10h+ar2]
                sub     esi, eax
                lea     ecx, ds:0[edx*4]
                cmp     esi, ecx
                jb      loc_143

loc_55:                                 ; CODE XREF: f(int,int *,int *,int *)+34
                cmp     eax, [esp+10h+ar1]
                jbe     short loc_67
                mov     esi, [esp+10h+ar1]
                sub     esi, eax
                neg     esi
```

[65] More about Intel C++ automatic vectorization: Excerpt: Effective Automatic Vectorization

```
                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 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 [66].

`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 piece 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[67], 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]
```

[66] More about data aligning: Wikipedia: data structure alignment

[67] More about GCC vectorization support: http://gcc.gnu.org/projects/tree-ssa/vectorization.html

```
loc_80484C8:                            ; CODE XREF: f(int,int *,int *,int *)+36
                mov     edx, [edi+eax*4]
                add     edx, [esi+eax*4]
                mov     [ebx+eax*4], edx
                add     eax, 1
                cmp     eax, ecx
                jnz     short loc_80484C8

loc_80484D8:                            ; CODE XREF: f(int,int *,int *,int *)+17
                                        ; f(int,int *,int *,int *)+A5
                add     esp, 0Ch
                xor     eax, eax
                pop     ebx
                pop     esi
                pop     edi
                pop     ebp
                retn
; ---------------------------------------------------------------------------
                align 8

loc_80484E8:                            ; CODE XREF: f(int,int *,int *,int *)+1F
                test    bl, 0Fh
                jnz     short loc_80484C1
                lea     edx, [esi+10h]
                cmp     ebx, edx
                jbe     loc_8048578

loc_80484F8:                            ; CODE XREF: f(int,int *,int *,int *)+E0
                lea     edx, [edi+10h]
                cmp     ebx, edx
                ja      short loc_8048503
                cmp     edi, eax
                jbe     short loc_80484C1

loc_8048503:                            ; CODE XREF: f(int,int *,int *,int *)+5D
                mov     eax, ecx
                shr     eax, 2
                mov     [ebp+var_14], eax
                shl     eax, 2
                test    eax, eax
                mov     [ebp+var_10], eax
                jz      short loc_8048547
                mov     [ebp+var_18], ecx
                mov     ecx, [ebp+var_14]
                xor     eax, eax
                xor     edx, edx
                nop

loc_8048520:                            ; CODE XREF: f(int,int *,int *,int *)+9B
                movdqu  xmm1, xmmword ptr [edi+eax]
                movdqu  xmm0, xmmword ptr [esi+eax]
                add     edx, 1
                paddd   xmm0, xmm1
                movdqa  xmmword ptr [ebx+eax], xmm0
                add     eax, 10h
                cmp     edx, ecx
                jb      short loc_8048520
                mov     ecx, [ebp+var_18]
                mov     eax, [ebp+var_10]
                cmp     ecx, eax
                jz      short loc_80484D8

loc_8048547:                            ; CODE XREF: f(int,int *,int *,int *)+73
                lea     edx, ds:0[eax*4]
                add     esi, edx
                add     edi, edx
```

```
                add     ebx, edx
                lea     esi, [esi+0]

loc_8048558:                            ; CODE XREF: f(int,int *,int *,int *)+CC
                mov     edx, [edi]
                add     eax, 1
                add     edi, 4
                add     edx, [esi]
                add     esi, 4
                mov     [ebx], edx
                add     ebx, 4
                cmp     ecx, eax
                jg      short loc_8048558
                add     esp, 0Ch
                xor     eax, eax
                pop     ebx
                pop     esi
                pop     edi
                pop     ebp
                retn
; ---------------------------------------------------------------------------

loc_8048578:                            ; CODE XREF: f(int,int *,int *,int *)+52
                cmp     eax, esi
                jnb     loc_80484C1
                jmp     loc_80484F8
_Z1fiPiS_S_     endp
```

Almost the same, however, not as meticulously as Intel C++ doing it.

2.18.2 SIMD `strlen()` () implementation

It is possible to implement `strlen()` function[68] 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);
    };
```

[68] 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 problem:

In Windows NT line of operation systems, but not limited to it, memory allocated by pages of 4 KiB (4096 bytes). Each win32-process have 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[69].

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`.

`pmovmskb eax, xmm1`

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

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[70]to LSB[71]):

```
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=0010000000100000b
```

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

[70]most significant bit

[71]least significant bit

2.19 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 3.5.3. 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 3.5.

- 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*[72]. 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 2.18):

```
/*
 * Generated S-box files.
 *
 * This software may be modified, redistributed, and used for any purpose,
 * so long as its origin is acknowledged.
 *
 * Produced by Matthew Kwan - March 1998
 */

#ifdef _WIN64
#define DES_type unsigned __int64
#else
#define DES_type unsigned int
#endif

void
s1 (
    DES_type    a1,
    DES_type    a2,
    DES_type    a3,
    DES_type    a4,
    DES_type    a5,
    DES_type    a6,
    DES_type    *out1,
    DES_type    *out2,
    DES_type    *out3,
    DES_type    *out4
) {
    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;
```

[72] assigning variables to registers

```
    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:

```
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:

```
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.

Chapter 3

Couple things to add

3.1 LEA instruction

`LEA` (*Load Effective Address*) is instruction intended not for values summing but for address forming, for example, 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

3.2 Function prologue and epilogue

Function prologue is instructions at function start. It is often something like this:

```
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 handy.

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 tree node. 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

3.3 npad

It's an assembler macro for label aligning by some specific border.

That's often need for the busy labels to where control flow is often passed, for example, loop 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 at all, 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
```

3.4 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 2.7 are present both for signed (for example, `JG`, `JL`) and unsigned (`JA`, `JBE`) operations.

3.4.1 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

3.5 Arguments passing methods (calling conventions)

3.5.1 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.

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

3.5.2 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.

```
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), do not have 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.

3.5.3 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, different compilers may do it differently. Of course, if you have two DLLs, one use another, and they are built by different compilers with *fastcall* calling conventions, there will be a problems.

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*.

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

function:
.. do something ..
```

[4]Size of *int* type variable is 4 in x86 systems and 8 in x64 systems

```
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.

3.5.4 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.

3.5.5 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).

3.5.6 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

Chapter 4

Finding important/interesting stuff in the code

Minimalism it's not a significant feature of modern software.

But not because programmers wrote 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.

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 6 disassembler can 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).

http://www.google.com/search?q=CsDecomprLZC

Astoundingly, MaxDB and SAP 6.0 software shared the same code for network packets compression/decompression.

4.1 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://blogs.conus.info/node/44.

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

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

4.2 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 6 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.

4.3 Constants

Some algorithms, especially cryptographical, use distinct constants, which is easy to find in code using IDA 6.

For example, MD5[3] 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.

4.3.1 Magic numbers

A lot of file formats defining a standard file header where *magic number*[4]is used.

For example, all Win32 and MS-DOS executables are started with two characters "MZ"[5].

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.

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

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

[5] 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()`:

```
.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:

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

And:

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

4.4 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 6, 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 6.0.1 (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:

```
gt -l:excel.exe bpx=excel.exe!base+0x11E91B,set(st0,666)
```

```
PID=36540|TID=24056|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00680006 EBX=0x00680018 ECX=0x00000001 EDX=0x00000001
ESI=0x00680000 EDI=0x00395404 EBP=0x0290FD9C ESP=0x0290FD58
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
Set ST0 register to 666.000000
```

Excel showing 666 in that cell what finally convincing us we find the right place.

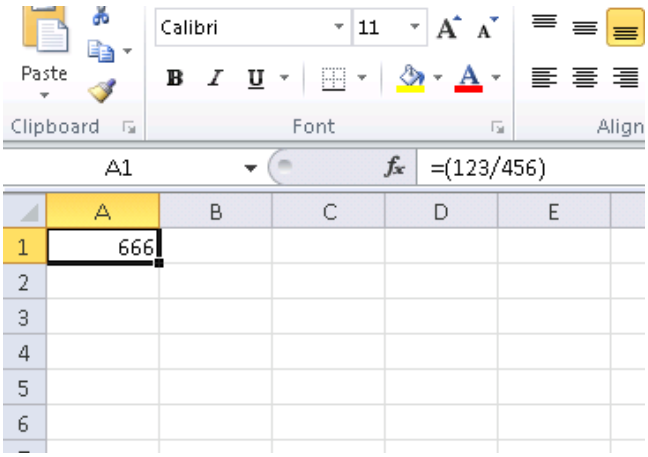

Figure 4.1: Practical joke worked

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.

```
gt64.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).

4.5 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 assembler. If you spot these, it can be said, with a big probability, this piece of code is hand-written.

Chapter 5

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 brochure.

5.1 Easy level

5.1.1 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
```

5.1.2 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
```

5.1.3 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
```

5.1.4 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
```

5.1.5 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
```

5.1.6 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
```

5.1.7 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
```

5.1.8 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
```

5.1.9 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
```

5.1.10 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;
};
```

5.2 Middle level

5.2.1 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
```

5.3 crackme / keygenme

/IFRUНесколько моих keygenme[2]: Couple of my keygenmes[3]:
http://crackmes.de/users/yonkie/

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

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

Chapter 6

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] /IFRUдля мелкой модификации кода в исполняемых файлах for small modifications of code in binary files.

6.0.1 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, allow 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 7

Books/blogs worth reading

7.1 Books

7.1.1 Windows

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

7.1.2 C/C++

- C++ language standard: ISO/IEC 14882:2003[2]

7.1.3 x86 / x86-64

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

7.2 Blogs

7.2.1 Windows

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

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

Chapter 8

Other things

8.1 More examples

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

8.2 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.

For example, 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
```

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.

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,
        0x05, 0x85, 0x45, 0xc5, 0x25, 0xa5, 0x65, 0xe5,
```

[2]Tiny Encryption Algorithm

```
        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 type *double* 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 type *double* 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 2.17, 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;
            }
```

```
                }
            }
        }
    }
```