C/C++ programming language notes

Dennis Yurichev
<dennis@yurichev.com>

Text version (March 17, 2017).
There is probably a newer version of this text, and also Russian language version also accessible at
http://yurichev.com/C-book.html
You may also subscribe to my twitter, to get information about updates of this text, etc: @yurichev, or to subscribe to mailing list.

Contents

[1] Operating System

Preface

Today, in year 2013, if one wants to write 1) as fast program as possible; 2) or as compact as possible for embedded systems or low-cost microcontrollers, the choice is very limited: C, C++ or assembly language. And as it seems in the near future, there are no alternative to these old but popular programming languages.

“Pure C” should be still considered, a huge number of large programs are still developed in it, e.g. Linux kernel, Windows NT OS line kernels, Oracle RDBMS, etc.

0.1 Target audience

This notes collections is not intended for beginners, neither for experts, it is rather for those who wants to fresh their C/C++ knowledge.

0.2 Thanks

Andrey ”herm1t” Baranovich, Slava ”Avid” Kazakov, Tuta Muniz, Shell Rocket, Daniel Craig.

Chapter 1

Common for C and C++

1.1 Header files

Header files are an interface description, a documentation in some sense. It is very convenient to work with the code in one editor window while having header files in another window for reference. It can be said it is advisable to make header files looks like references.

1.2 Definitions and declarations

The difference between *declarations* and *definitions* is:

- *Declaration* declares name/type of variable or name and argument types and also returning value type of function or method. Declarations are usually enlisted in the header .h or .hpp-files, so the compiler while compiling individual .c or .cpp-file may have an access to the information about all names and types of external (usually global) variables and functions/methods.

  Data types are also *declared*.

- *Definition* defines a value of (usually global) variable or a body of function/method. It is usually take place in the individual .c or .cpp-file.

1.2.1 C/C++ declarations

Local variable declarations

It was possible to declare variables only at the function beginning in C. And anywhere in C++.

It was also not possible to declare counter or Iterator in for() (it was possible in C++):

```
for(int i=0; i<10; i++)
        ...
```

The new C99(2.5.10) standard allows this.

static If the global variables (or functions) are declared as *static*, its scope is limited by current file. However, local variables inside a function may also be declared as *static*, then this variable will be global instead of local, but its scope will be limited by a function.

For example:

```
void fn(...)
{
        for(int x=0; x<100; x++)
        {
                static int times_executed = 0;
                times_executed++;
        }
};
```

For example, it might be helpful for the `strtok()` implementing, because this function should store something between calls.

forward declaration

As it is well-known, in the header files (headers) function declarations are usually present, i.e., function names, arguments and types, returning type, but no function bodies. This is done for the compiler so it will have information, what it is working with, without delving into the intricacies of function implementations.

The same can be done for types. In order not to include with the help of #include the file with a class definitions into the other header file, one can just declare the type presence.

For example, you work with complex numbers and you have a such structure somewhere:

```
struct complex
{
        double real;
        double imag;
};
```

And let's say it is declared in the file my_complex.h.

Of course, one should include the file if one have intention to work with variables of *complex* type and specific structure fields. But if you declare your functions using the structure in other header file, you may not include my_complex.h there, compiler just needs to know that the *complex* is a structre:

```
struct complex;

void sum(struct complex *x, struct complex *y, struct complex *out);
void pow(struct complex *x, struct complex *y, struct complex *out);
```

This may speed up the compilation process and also solve circular dependencies, when two modules uses functions and type of each other.

Frequent caveats

In order to declare two pointers to *char*, one may write by inertion:

```
char* p1, p2;
```

It is not correct, because the compiler treat this declaration as:

```
char *p1, p2;
```

... and declares the pointer to *char* and just *char*.

This one is correct:

```
char *p1, *p2;
```

const

To declare variables, function arguments and C++ class methods as const is advisable because:

- Self-documentation — it can be easy seen visually that the element is read-only.
- Protection from errors: in case of global const-variable, the process will crash while attempting to write to it due to memory protection. The compiler also reports an error if to try to modify a const-argument inside of a function.
- Optimization: the compiler considering the element is always read-only, may generate faster code for using it.

It is highly advisable to declare all function arguments which you do not plan to modify as const. For example, the `strcmp()` function is changing nothing in the input arguments, so they are both usually declared as const. The strcat() changing nothing in the second argument, but changing something in the first, so it is usually declared with a *const* in the second argument.

C++ In the C++, a class methods which are not changing anything in the object is highly advisable to be declared as *const*. const-methods of a class are also called as *accessors*, while non-const methods as *manipulators* [11].

Datatypes

long double *float* is 32-bit number in IEEE 754 format, *double* is 64-bit variable but x86 FPU-coprocessor is in fact operating 80-bit numbers. There is another type for those: *long double*, it is supported in the GCC[1] but not in MSVC[2].

int The *int* usually occupies the same number of bits as general purpose CPU registers. However, in the x86-64, for better backward compatibility, *int* width is still 32 bits.

short, long and long long At least in the MSVC and GCC *short* 16-bit, *long* — 32-bit, and *long long* — 64-bit.

In order to avoid confusion, *stdint.h* (at least in C99) has new types: *int8_t*, *uint8_t*, *int16_t*, *uint16_t*, *int32_t*, *uint32_t*, *int64_t*, *uint64_t*.

bool bool is present in the C++, but also in the C starting at C99(2.5.10) (stdbool.h).

Both in the MSVC and GCC, bool occupies 1 byte.

There is a synonymous to the *int* type in Windows API — BOOL.

Signed or unsigned? Signed types (int, char) are used much more often than unsigned (unsigned int, unsigned char) [3].

However, in the sense of the code self-documenting, if you declare a variable which will not be assigned to a negative value, including array indices, perhaps, unsigned type is better. For example, *unsigned* type is often used in LLVM at the places where *int* might be used.

If you work with bytes, e.g. with bytes in memory, then perhaps *unsigned char* is better.

Aside from that, this may help protecting from the errors related to integer overflow [1].

As a simple example:

```
#define MAX_BUFFER_SIZE 1024

void f(int size)
{
        if (size>MAX_BUFFER_SIZE)
                die ("Too large!");
        void *p=malloc (size);
        ...
};
```

If *size* will be, for example, -1, then malloc() will be called with an argument `0xffffffff` (it is 4294967295). Of course, we need to add a second sanitizing check: *if (size<0)*, but such check here will have absurdical look.

So, the type *unsigned*should be used here, maybe even *size_t*. *size_t* declares a big enough type able to store a size of any, big enough memory block. It is *unsigned int* on 32-bit architectures, and *unsigned int64* on 64-bit ones.

char or uint8_t instead of int? One may think that is a value will always be in $0..100$ limits, then it is not necessary to allocate the whole 32-bit *int*, smaller types may be enough like *char* or *unsigned char*. Besides, it will require less memory.

It is not so. Because of aligning by 4-bytes border (or by 8-bytes border in 64-bit architectures), the variables declared with the type *char*, requires as much space as *int*.

Of course the compiler may allocate only 1 bytes for the *char*, but then CPU[4] will spent more time for accessing "unaligned" by border bytes.

Specific bytes processing may be more "expensive" and slower then processing 32-bit or 64-bit values because CPU registers are usually has the same width as CPU bits. Even more than that, RISC[5]-units (e.g. ARM) may not be able to work with specific bytes internally at all because they have only 32-bit registers.

So if you considering about type for the local variable, *int/unsigned int* may be better.

On the other hand, which types are better suited for a structures? This is a question of seeking balance between speed and compactness. On the one hand, one may use *char* for a small variables, flags, bitfields, enums, etc, but one should not forget that access to these variables will be slower. On the other hand, if to assign *int* to each variables, working with a structure will be faster, but it will require more space in memory.

For example:

[1] GNU Compiler Collection

[2] Microsoft Visual C++

[3] The data type of *char* is not fixed in the standard, but in GCC and MSVC it is exactly signed type by default. In can be changed in the GCC by adding key `-funsigned-char` and in MSVC key `/J`.

[4] Central processing unit

[5] Reduced instruction set computing

```
struct
{
        char some_flags; // 1 byte
        void* ptr; // 4/8 bytes, offset: +1
} s;
```

If to compile this with structure packing by 1-byte border, access to the *some_flags* in the memory may be even faster then to *ptr*, because the first field is aligned by 4-byte border, while the second is not.

If to compile this by default structure packing, then 4 bytes will be allocated for the first field and the offset of the second field will be +4.

Summarizing: if compactness and memory footprint is important, then *char*, *uint16_t*, etc, may be used.

x86-64 or AMD64 On the new 64-bit x86 CPUs, the *int/unsigned int* is still 32-bit, perhaps, for compatibility. So if one need 64-bit variables, one may use *uint64_t* or *int64_t*.

But pointers, of course, has 64-bit width.

typedef

`typedef` introduces *synonym* for a data type. It is often used for structures, for the reason not to write *struct* each time before its name, e.g.:

```
typedef struct _node
{
        node *prev;
        node *next;
        void *data;
} node;
```

A lot of such examples may be found in header files in the Windows SDK (Windows API).

Nevertheless, *typedef* can be also used not only for structures, but also for usual integral types like:

```
typedef int age;
int compute_mean (age wife, age husband);

typedef int coord;
void draw (coord X, coord Y, coord Z);

typedef uint32_t address;
void write_memory (address a, size_t size, byte *buf);
```

As we can see, *typedef* here may help with code documentation, it is now easier to read.

For example, the *time_t* (The time in the UNIX time format, e.g. what the localtime() returns), it is in fact 32-bit number, but the type is defined in the *time.h* file usually as:

```
typedef long __time32_t;
```

A preprocessor directive `#define` may be used here (many do so), but it is worse in the sense of errors handling during compilation.

typedef criticism Nevertheless, such well-known and experiences programmer as Linus Torvalds is against typedef usage: [17].

1.2.2 Definitions

String declarations

Character sequences used in strings

\0	0x00	zero byte
\a	0x07	bell [6]
\t	0x09	tabulation
\n	0x0A	line feed (LF)
\r	0x0D	carriage return (CR)

The difference between LF and CR is that in old dot-matrix printers LF mean line feed by one line down, and CR carriage return to the left margin of paper. So both characters must be transmitted to the printer.

Outputting CR without LF allows to rewrite current string in the console:

```
for (;;)
{
        // do something
        // how much we processed?
        percents=ammount_of_work/total_work*100;
        printf ("%d%% complete\r", percents);
};
```

This is often used in the file archivers for displaying current status and wget.

In C and UNIX LF is traditionally accepted as newline symbol.

In the DOS and then Windows — CR+LF.

The string defined as multi-string Not widely known C feature, long strings can be defined as:

```
const char* long_line="line 1"
        "line 2"
        "line 3"
        "line 4"
        "line 5";

...

printf ("Some Utility v0.1\n"
        "Usage: %s parameters\n"
        "\n"
        "Authors:...\n", argv[0]);
```

It is somewhat resembling “here document”[7] in UNIX-shells and Perl.

1.3 Language elements

1.3.1 Comments

It is sometimes useful to insert them right into a function call, in order to have a visual note about meaning of an argument:

```
f (val1, /* a very special flag! */ false, /* another special flag here */ true);
```

The whole code block can be commented with the help of #if [8]:

```
        ta      = aemif_calc_rate(t->ta, clkrate, TA_MAX);
        rhold   = aemif_calc_rate(t->rhold, clkrate, RHOLD_MAX);
#if 0
        rstrobe = aemif_calc_rate(t->rstrobe, clkrate, RSTROBE_MAX);
        rsetup  = aemif_calc_rate(t->rsetup, clkrate, RSETUP_MAX);
        whold   = aemif_calc_rate(t->whold, clkrate, WHOLD_MAX);
#endif
        wstrobe = aemif_calc_rate(t->wstrobe, clkrate, WSTROBE_MAX);
        wsetup  = aemif_calc_rate(t->wsetup, clkrate, WSETUP_MAX);
```

This might be more convenient then usual way because the text editor or IDE[9] in this case will not “break” indentation while auto-indentation.

[7] https://en.wikipedia.org/wiki/Here_document

[8] preprocessor directive

[9] Integrated development environment

1.3.2 goto

Usage of *goto*[10] is considered as bad taste and harmful [4] [3], nevertheless, its usage in reasonable doses [9] may be very helpful.

One frequent example is return from a function:

```
void f(...)
{
        byte* buf1=malloc(...);
        byte* buf2=malloc(...);

        ...

        if (something_goes_wrong_1)
                goto cleanup_and_exit;

        ...

        if (something_goes_wrong_2)
                goto cleanup_and_exit;

        ...

cleanup_and_exit:
        free(buf1);
        free(buf2);
        return;
};
```

More complex example:

```
void f(...)
{
        byte* buf1=malloc(...);
        byte* buf2=malloc(...);

        FILE* f=fopen(...);
        if (f==NULL)
                goto cleanup_and_exit;

        ...

        if (something_goes_wrong_1)
                goto close_file_cleanup_and_exit;

        ...

        if (something_goes_wrong_2)
                goto close_file_cleanup_and_exit;

        ...

close_file_cleanup_and_exit:
        fclose(f);

cleanup_and_exit:
        free(buf1);
        free(buf2);
        return;
};
```

[10] statement

If to remove all *goto* in these examples, one will need to call *free()* and *fclose()* before each return from the function (*return*)which adds a lot of mess.

Usage of *goto* is, for example, approved in [16].

1.3.3 for

The for() statement, as we know, has 3 expressions: 1st computing before all iterations begin, 2nd computing before each iteration and the 3rd — after each iteration.

And of course, there might be written something different from the usual counter.

Caveat #1

If to write this:

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

void count_spaces(char *s)
{
        int spaces=0;

        for (int i=0; i<strlen(s); i++)
        {
                if (s[i]==' ')
                        spaces++;
        };

        printf ("spaces=%d\n", spaces);
};

int main()
{
        count_spaces("The quick brown fox jumps over the lazy dog");
        return 0;
};
```

... perhaps this is a mistake: `strlen(s)` will be called before each iteration — that is the code MSVC 2010 generated. However, GCC 4.8.1 calls `strlen(s)` only once, at the loop beginning.

Comma

Comma [6, 6.5.17] — is not widely understood C feature, however, it is very useful for using in a for() declarations.

For example, it is useful to have two counter or iterators simultaneously. Let the counter just counts from 0 adding 1 at each iteration, and the Iterator points to the list element:

```
#include <iostream>
#include <list>

int main()
{
        std::list<int> l;

        l.push_back(123);
        l.push_back(456);
        l.push_back(789);
        l.push_back(1);

        int i;
        std::list<int>::iterator it;
        for (i=0, it=l.begin(); it!=l.end(); i++, it++)
                std::cout << i << ": " << *it << std::endl;
```

```
        return 0;
};
```

This will dumps predictable result:

```
0: 123
1: 456
2: 789
3: 1
```

However, it is not possible to declare iterators with its different types in for() clause:

```
        for (int i=0, std::list<int>::iterator it=l.begin(); it!=l.end(); i++, it++)
```

Nevertheless, variables of the same type can be defined:

```
        for (int i=0, j=10; i<20; i++, j++)
```

continue

continue is unconditional goto to the end of loop body.

This may be very useful, for example, in such code:

```
for (...)
{
        if (is_element_satisfied_criteria_1(...)==true)
        {
                // do something need in is_element_satisfied_criteria_2()

                if (is_element_satisfied_criteria_2(...)==true)
                {
                        do_something_1();
                        do_something_2();
                        do_something_3();
                };

        };
};
```

... it is all can be replaced by neat:

```
for (...)
{
        if (is_element_satisfied_criteria_1(...)==false)
                continue;

        // do something need in is_element_satisfied_criteria_2()

        if (is_element_satisfied_criteria_2(...)==false)
                continue;

        do_something_1();
        do_something_2();
        do_something_3();
};
```

1.3.4 if

Instead short *if* the shorter ?: clause is advisable, e.g.:

```
char* get_name (struct data *s)
{
        return s->name==NULL ? "<name_unknown>" : s->name;
```

```
};

...

printf ("val=%s\n", val ? "true" : "false");
```

C++: variable declarations in if()

It is avaiable at least in C++03 standard:

```
if (int a=fn(...))
{
     ...
     cout << a;
     ...
};
```

They can be declared likewise also in switch().

1.3.5 switch

It is sometimes boring to write the same again and again:

```
switch(...)
{
        case 0:
        case 1:
        case 2:
        case 3:
                fn1();
                break;
        case 4:
        case 5:
        case 6:
        case 7:
                fn2();
                break;
};
```

And this non-standard GCC extension [11] may make things somewhat simpler:

```
switch(...)
{
        case 0 ... 3:
                fn1();
                break;
        case 4 ... 7:
                fn2();
                break;
};
```

So if you plan to use only GCC compiler, it is possible to do so.

Variable declarations inside switch()

It is not possible, but it is possible to open a new block and to declare them there (in C++ or starting from C99):

```
switch(...)
{
        case 0:
                {
```

[11] http://gcc.gnu.org/onlinedocs/gcc/Case-Ranges.html

```
                        int x=1,y=2;
                        fn1(x, y);
                };
                break;
        case 1:
        case 2:

        ...
};
```

1.3.6 sizeof

Usually, sizeof() is applied to integral types or to structures, but nevertheless it is possible to apply it to arrays as well:

```
        char buf[1024];
        snprintf(buf, sizeof(buf), "...");
```

Otherwise, if to specify array length (1024) in both places (in `buf` declaration and as a second argument of `snprintf()`), then the value is have to be changed at the both places each time, and it is easy to forget about this.

If one need wide-strings, then sizeof() can be applied to *wchar_t* (which is in turn, 16-bit data type *short*):

```
        wchar_t buf[1024];
        swprintf(buf, sizeof(buf)/sizeof(wchar_t), "...");
```

sizeof() returns the size in bytes, so it will be here $1024 * 2$, i.e., 2048. But we can divide this value by length of one array element (*wchar_t*) is 2 in bytes, in order to get elements number in array (1024).

sizeof() can be applied to array of structures:

```
struct phonebook_entry
{
        char *name;
        char *surname;
        char *tel;
};

struct phonebook_entry phonebook[]=
{
        { "Kirk", "Hammett", "555-1234" },
        { "Lars", "Ulrich", "555-5678" },
        { "James", "Hetfield", "555-1122" },
        { "Robert", "Trujillo", "555-7788" }
};

void dump (struct phonebook_entry* input)
{
        for (int i=0; i<sizeof(phonebook)/sizeof(struct phonebook_entry); i++)
                printf ("%s %s - %s\n", input[i].name, input[i].surname, input[i].tel);
};
```

sizeof(phonebook) — is a size of the whole array of structures in bytes. `sizeof(struct phonebook_entry)` — is a size of one structure in bytes. By division we get number of structures in an array.

1.3.7 Pointers

As Donald Knuth once said in the interview [10], the way C handles pointers, was a brilliant innovation at the time.

So let us fix terminology. A pointer is a just an address of some element in memory. The reason pointers are so popular is that an address of object is much easier to pass into a function instead of passing the whole object — because it is absurdly.

Besides, calling function, e.g. processing a data array, will just change something in it instead of returning new one, which is absurdly too.

Let's take a simple example. The standard C function *strtok()* just divide string by substrings using specified character as delimiter. For example, we may specify the string `The quick brown fox jumps over the lazy dog` and set the space as a delimiter.

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

int main()
{
        char str[] = "The quick brown fox jumps over the lazy dog"; // correct
        //char *str= "The quick brown fox jumps over the lazy dog"; // incorrect
        char *sep = " ";

        /* get the first token */
        char *token = strtok(str, sep);

        /* walk through other tokens */
        while( token != NULL )
        {
                printf( "%s\n", token );
                token = strtok(NULL, sep);
        }
};
```

What we got on output:

```
The
quick
brown
fox
jumps
over
the
lazy
dog
```

What is going on here is that the *strtok()* just searching for the next space in the input string (or any other delimiter set), writes 0 to it (this is string terminator by C conventions) and returns a pointer to that place.

As a shortcoming, it can be said that the *strtok()* function "garbles" input string, writing zeros at the delimiter's places.

What is worth to note: no strings or substrings copied in memory. The input string is still on its own place.

It is only pointer to the string (or its address) is passed to the *strtok()* function.

The function then after it writes 0, returns *address* of each consecutive "word".

The address of the "word" is then passed to the *printf()*, where it dumped to the console.

N.B. An incorrect declaration of *str* is present in the source code.

It is incorrect in that sense that the C string has type *const char**, i.e., it is located in the constant data segment, write-protected.

If do so, then the *strtok()* will not be able modify the input string by writing zeros there and the process will crash.

So, in our example, the string is allocated as an array of *char* instead of array of *const char*.

Generalizing, we may say all standard C strings functions works with them using only their addresses.

For example, the function of string comparison *strcmp()* takes addresses of two strings and compare them by one character. It would be absurdly to copy these strings to some other place so the *strcmp()* may process them.

The difficulty of C pointers understanding is in the fact that pointer is a "part" of an object. The pointer to the string is not the string itselfs. The string should be placed somewhere in memory, a memory should be allocated for it before, etc.

In a higher level PL[12] an object and a pointer may be represented as a single whole, and that is makes understanding simpler.

It is however not mean that a strings and other objects are copied misspendinly in these PL — a pointers are used there internally likewise as in C, but this mechanisms are hidden from the programmer.

Passing a value to a function is also called "call by value" or "pass by value" while passing a pointer to an object is called "call by reference" or "pass by reference".

Syntactic sugar for array[index]

For the sake of simplification, it could be said that C has not arrays at all, it has only syntactic sugar for expressions like *array[index]*.

[12] Programming Language

For example, perhaps you saw this trick:

```
printf ("%c", 3["hello"]);
```

It outputs 'l'.

This happens because the expression *a[i]*, is in fact translating into **(a+i)* [6, 6.5.2/1]. *3["hello"]* is translated into **(3+"hello")*, and *"hello"* is just a pointer to array of characters like *const char**.

3+"hello" as a result is a pointer to the part of string, *"lo"*. And **("lo")* is a 'l'. That is why it works.

It is not advisable to write such things unless your intentions is to participate in IOCCC[13][14]. So I demostrated the trick here in order to explain that the *a[i]* is a syntactic sugar.

With some persistence, it is possible not to use indexed arrays in C at all, but it will not be very æsthetical though.

By the way, now it is easy to understand how negative array indices works. *a[-3]* is translating into **(a-3)*, and that is how the element before array itself is addressed. Despite it is possible, one should use this only if one exactly knows what one does.

Another array negative indices trick: for example, if you used to address arrays starting not from 0 but from 1 (like in FORTRAN), then you may do something like this:

```
void f (int *a)
{
        a[1]=...; // first element
        a[2]=...; // second element
};

int main()
{
        int array[10];
        f(&array[-1]); // passing a pointer to the one int element before array
};
```

But again it is hard to say if the trick is justified.

So C array in some sense is just a memory block plus a pointer to it.

That is why array name in C may be treated as a pointer:

If to declare global variable *int a[10]*, then *(a)* will have the type *int**. When further the following expression will be appeared: *x=a[5]*, the expression will be translated into *x=*(a+5)*. From the array start (i.e., from the first array element) 5 elements will be counted, then the element will be read from that point for the storing it into *(x)*.

Pointer arithmetic

Simple example:

```
#include <stdio.h>

struct phonebook_entry
{
        char *name;
        char *surname;
        char *tel;
};

struct phonebook_entry phonebook[]=
{
        { "Kirk", "Hammett", "555-1234" },
        { "Lars", "Ulrich", "555-5678" },
        { "James", "Hetfield", "555-1122" },
        { "Robert", "Trujillo", "555-7788" },
        { NULL, NULL, NULL }
};

void dump1 (struct phonebook_entry* input)
{
        for (int i=0; input[i].name; i++)
```

[13] The International Obfuscated C Code Contest

[14] http://www.ioccc.org/

```
                printf ("%s %s - %s\n", input[i].name, input[i].surname, input[i].tel);
};

void dump2 (struct phonebook_entry* input)
{
        for (struct phonebook_entry* i=input; i->name; i++)
                printf ("%s %s - %s\n", i->name, i->surname, i->tel);
};

void main()
{
        dump1(phonebook);
        dump2(phonebook);
};
```

We define a global array of structures. If to compile this in GCC with a *-S* key or in MSVC with a */Fa* key, we will see in assembly language listing how the compiler placed these strings.

The compiler placed them as a linear array of string pointers, that is how:

cell 0	string address “Kirk”
cell 1	string address “Hammett”
cell 2	string address “555-1234”
cell 3	string address “Lars”
cell 4	string address “Ulrich”
cell 5	string address “555-5678”
cell 6	string address “James”
cell 7	string address “Hetfield”
cell 8	string address “555-1122”
cell 9	string address “Robert”
cell 10	string address “Trujillo”
cell 11	string address “555-7788”
cell 12	0
cell 13	0
cell 14	0

The functions *dump1()* and *dump2()* are equivalent to each other.

But in the first function counter *(i)* is beginning at 0 and 1 is added to it at each iteration.

In the second function Iterator *(i)* points to the beginning of the array and then size of structure is added to it (instead of 1 byte, how one can mistakenly think), this mean, at each iteration, *(i)* points to the next element of array.

Pointer to functions

Often used for callbacks.

The address of function can be set directly, so it is possible to jump to an arbitrary address, it is useful in embedded-programming:

```
void (*func_ptr)(void) = (void (*)(void))0x12345678;
func_ptr();
```

However, it should be noted that this is not the same thing as unconditional jump, because return address is saved in the stack, maybe something else.

1.3.8 Operators

==

Somewhat unpleasant mistakes may appear if in *if(a==3)* condition become *if(a=3)* in result of typo. Because the statement *a=3* “returns” 3, and 3 is not a 0, so the *if()* condition will always trigger.

It was fashionable in past to protect from such mistakes by writing: *if(3==a)*, and thus, we will get a *if(3=a)* in case of typo and the compiler will report error instantly.

Nevertheless, in modern times, compilers are usually warns if to write *if(a=3)*, so elements swapping in conditions is probably not necessary these days.

Short-circuit evaluation and operator precedence artefact

Let's see what is *short-circuit evaluation*.

It is when in the expression *if(a && b && c)*, the part *(b)* will be calculated only if *(a)* — is true, and *(c)* will be calculated only if *(a)* and *(b)* — are both true. and they will be computed exactly in the same order as specified.

Sometimes we can see expression like: *if (p!=NULL && p->field==123)* — and this is completely correct. The field *field* in the structure to which *(p)* points will be computed only if the pointer *(p)* not equals to *NULL*.

The same story about "or", if in the expression *if (a || b || c)* subexpression *(a)* will be "true", others will not be computed.

It is useful when one need to call several functions: *if (get_flagsA() || get_flagsB() || get_flagsC())* — if first or second will return *true*, others will not be called at all.

This feature is not unique for C/C++ [15].

Some time ago [14], there was no operators *&&* and || in B and BCPL (C precursors), but in order to implement *short-circuit evaluation* in them, the priority of the operators *&* and | was made higher than in ^ or + [16].

That allowed to write something like *if (a==1 & b==c)* while using *&* instead of *&&*. That is where that artefact came from.

So one often mistake is to forget about higher priority of these operators and to write e.g., *if (a&1==0)*, which should be taken in brackets: *if ((a&1)==0)*.

! and ~

~ (tilde) is a bitwise inversion of all bits in a value.

The operation is often used for function results invertion. For example, strcmp() in case of strings equivalence, returns 0. So we can write:

```
if (!strcmp(str1, str2))
{
        // do something in case of strings equivalence
};
```

... instead of `if (strcmp (...)==0)`.

Also, two consecutive exclamation points can be used for transforming any value into *bool* type: 0 — false (0); not zero — true (1).

For example:

```
bool some_object_present=!!struct->object;
```

Or:

```
#define FLAG 0x00001000
bool FLAG_present=!!(value & FLAG);
```

And also:

```
bool bit_7_set=!!(value & (1<<7));
```

1.3.9 Arrays

In C99(2.5.10) it is possible to pass array in the function arguments.

Strictly speaking, array of bytes can be passed in the older C standards, by encoding all bytes including zero in a string (let's determine if the byte (*c*) is present in a byte array)(2.2.5):

```
if (memchr ("\x12\x34\x56\x78\x00\xAB", c, 6))
        ...
```

The bytes after zero is encoded finely.

However, it is possible in C99 to pass an array of other types, like unsigned int:

```
unsigned int find_max_value (unsigned int *array, size_t array_size);

unsigned int max_value=find_max_value ((unsigned[]){ 0x123, 0x456, 0x789, 0xF00 }, 4);
```

[15] Here is a list of PL where *short-circuit evaluation* existhttps://en.wikipedia.org/wiki/Short-circuit_evaluation. It is not about C, but interesting nevertheless: if to write in bash *cmd1 && cmd2 && cmd3*, then each next command will be executed only if the previous was executed with success. It is also *short-circuit*.

[16] C++ Operator Precedence: http://en.cppreference.com/w/cpp/language/operator_precedence

Search for the element in the array can be implemented with the help of bsearch() or lfind()(2.5.7), search and insertion with the help of lsearch() [17].

Initialization

It is possible in GCC [18] to initialize array parts:

```
struct a
{
        int f1;
        int f2;
};

struct a tbl[8] =
{
[0x03] =        { 1,6 },
[0x07] =        { 5,2 }
};
```

... but it is non-standard extension.

1.3.10 struct

In the C99(2.5.10) it is possible to initialize specific structure fields. Fields not set will be filled by zeroes. A lot of such examples can be found in Linux kernel.

```
struct color
{
        int R;
        int G;
        int B;
};

struct color blue={ .B=255 };
```

And even more than that, it is possible to create a structure right in the function arguments, e.g.:

```
struct color
{
        int R;
        int G;
        int B;
};

void print_color_info (struct color *c)
{
        printf ("%d %d %d\n", c->R, c->G, c->B);
};

int main()
{
        print_color_info(&blue);
        print_color_info(&(struct color){ .G=255 });
};
```

The structure is also can be returned from the function in the same way:

```
struct pair
{
        int a;
        int b;
```

[17] works like lfind(), but if the element is absent there, it also inserts it

[18] http://gcc.gnu.org/onlinedocs/gcc/Designated-Inits.html

```
};

struct pair f1(int a, int b)
{
        return (struct pair) {.a=a, .b=b};
};
```

Read more about structures in the section “Object-oriented programming in C”(2.3.3).

Structure fields placement (cache locality)

In modern x86 CPUs (both Intel and AMD) a multi-level cache-memory present. The fastest cache-memory (L1) is divided by 64-byte elements (cache-lines) and any memory access resulting in filling a whole line [5].

It can be said that any memory access (on aligned boundary) fetches 64 bytes into cache at once.

So if a data structure is larger than 64 bytes, it is very important to divide it by 2 parts: the most demanded fields and the less ones. It is desirable to place the most demanded fields in the first 64 bytes.

C++ classes are also concerned.

1.3.11 union

`union` is often used when in some place of the structure one need to store various data types by choice. For example:

```
union
{
        int i; // 4 bytes
        float f; // 4 bytes
        double d; // 8 bytes
} u;
```

Such `union` allows to store one of these variables by choice. It will require the same ammount of space as a largest element (double) — 8 bytes.

`union` is often used as a way to access some data type as another data type.

For example, as we know, each XMM-register in SSE may be represented as 16 bytes, 8 16-bit words, 3 32-bit words, 2 64-bit words, 4 float variables and 2 double variables. This is how it can be declared:

```
union
{
        double d[2];
        float f[4];
        uint8_t b[16];
        uint16_t w[8];
        uint32_t i[4];
        uint64_t q[2];
} XMM_register;

union XMM_register reg;

reg.u.d[0]=123.4567;
reg.u.d[1]=89.12345;

// here we can use reg.u.b[...]
```

It is also handy to use it with a structure where fields has bit granularity. As x86-CPU flags:

```
typedef struct _s_EFLAGS
{
    unsigned CF : 1;
    unsigned reserved1 : 1;
    unsigned PF : 1;
    unsigned reserved2 : 1;
    unsigned AF : 1;
    unsigned reserved3 : 1;
    unsigned ZF : 1;
```

```
    unsigned SF : 1;
    unsigned TF : 1;
    unsigned IF : 1;
    unsigned DF : 1;
    unsigned OF : 1;
    unsigned IOPL : 2;
    unsigned NT : 1;
    unsigned reserved4 : 1;
    unsigned RF : 1;
    unsigned VM : 1;
    unsigned AC : 1;
    unsigned VIF : 1;
    unsigned VIP : 1;
    unsigned ID : 1;
} s_EFLAGS;

typedef union _u_EFLAGS
{
    uint32_t flags;
    s_EFLAGS s;
} u_EFLAGS;
```

Thus it possible to load flags as a 32-bit value into *flags* field and then, from the *(s)* field to access specific bits. Or conversely, to modify bits and then to read the *flags* field. A lot of such examples you may find in the Linux kernel.

One more example of *union* usage for determining current endianness:

```
int is_big_endian(void)
{
    union {
        uint32_t i;
        char c[4];
    } bint = {0x01020304};

    return bint.c[0] == 1;
}
```

[19]

tagged union

It is `union` plus flag (tag), defines type of `union`. For example, if we need a variable which can be a number, a float point number, a text string (as in dynamically typed PL [20]), then we may declare such structure:

```

enum var_type
{
        INT,
        DOUBLE,
        STRING
};

struct
{
        enum var_type tag; // 4 bytes
        union
        {
                int i; // 4 bytes
                double d; // 8 bytes
                char *string; // 4 bytes (on 32-bit architecture)
        } u;
```

[19] example is taken from: http://stackoverflow.com/questions/1001307/detecting-endianness-programmatically-in-a-c-program
[20] in Visual Basic it also called “variant type”

```
} variable;
```

The whole size of the structure is $8 + 4 = 12$ byte. It is much more compact than to allocate fields for the variable of each type.

Beginning with C11 [7], `(u)` may not be specified, it is called "anonymous union":

```
struct
{
        enum var_type tag; // 4 bytes
        union
        {
                int i; // 4 bytes
                double d; // 8 bytes
                char *string; // 4 bytes (on 32-bit architecture)
        };
} variable;
```

... and to access them as `variable.i`, `variable.d`, etc.

tagged pointers

Let's back to the example of variable declaration, which can be a number, a floating point number, and a text string. Largest type — double (8 bytes), this means, by storing a lot of such blocks in memory back to back, a pointer to each block will always be aligned on 8-byte border. Even more than that, glibc malloc() always allocates memory blocks by 8-byte border. Hence, the pointer to such block will always have zero in 3 lowest bits. And if so, these lowest bits may be used for something. One chance is to store there the type of *union*. We have only 3 variable types, so we need only 2 bits for storing a number in 0..2 range.

This is what is called *tagger pointer*. That is the way to make memory footprint smaller and to get rid of data type defining *enum* from the structure.

This approach is very popular in LISP-interpreters and compilers because LISP atoms is a similar variable defining structures, a there are may be a lot of them in memory, so it can reduce memory footprint by using lowest bits of pointers.

Likewise, any other information maybe stored in the tagger pointer bits.

As a negative side, one should always keep in mind that this is not an usual pointer, but has more information. Debuggers will not be able to work with such pointers correctly.

1.4 Preprocessor

The preprocessor handles directives started with # — #define, #include, #if, etc.

Listing 1.1: "or"

```
#if defined(LINUX) || defined(ANDROID)
```

1.5 Standard values for compilers and OS

- `_DEBUG` — debugging build.
- `NDEBUG` — non-debugging (release) build.
- `__linux__` — OS Linux.
- `_WIN32` — OS Windows. Present in both x86 and x64 builds. Absent in Cygwin.
- `_WIN64` — Present in x64 builds for OS Windows.
- `__cplusplus` — Present in a C++ projects.
- `_MSC_VER` — MSVC compiler.
- `__GNUC__` — GCC compiler.
- `__APPLE__` — compilation for Apple devices.
- `__arm__` — compilation for ARM processor (GCC, Keil).

- `__ppc__` — compilation for PowerPC 32-bit.
- `__ppc64__` — compilation for PowerPC 64-bit.
- `__LP64__` or `_LP64` — GCC: compilation for 64-bit architecture [21].

That is how it is possible to write various code pieces for various compilers and OS.
Other OS-related macros: http://sourceforge.net/p/predef/wiki/OperatingSystems/

1.5.1 More standard preprocessor macros

`__FILE__`, `__LINE__`, `__FUNCTION__` — current file name, current line number, current function name respectively.
In order to get values of `__FILE__` and `__FUNCTION__` in UTF-16, the following hack may be used:

```
#define CONCAT(x, y) x##y
#define WIDEN(x) CONCAT(L,x)

wprintf (L"%s\n", WIDEN(__FUNCTION__));
```

1.5.2 "Empty" macro

_DEBUG is well-known macro without any value. It is usually checked its presence or absence. Here is another example of useful "empty" macro:
In the Windows API header files we can find this:

```
typedef NTSTATUS
(NTAPI *TDI_REGISTER_CALLBACK)(
  IN PUNICODE_STRING DeviceName,
  OUT HANDLE *TdiHandle);

...

typedef NDIS_STATUS
(NTAPI *CM_CLOSE_CALL_HANDLER)(
  IN NDIS_HANDLE  CallMgrVcContext,
  IN NDIS_HANDLE  CallMgrPartyContext  OPTIONAL,
  IN PVOID  CloseData  OPTIONAL,
  IN UINT  Size  OPTIONAL);
```

IN, OUT and OPTIONAL — are "empty" macros defined as:

```
#ifndef IN
#define IN
#endif
#ifndef OUT
#define OUT
#endif
#ifndef OPTIONAL
#define OPTIONAL
#endif
```

They carry no information for compiler at all, they are intended for documenting purposes, to mark function arguments.

1.5.3 Frequent mistakes

#1

For example, you may want to define a macro for taking the power of a number:

```
#define square(x)      x*x
```

It is a mistake because the expression *square(a+b)* will "unfold" into $a + b * a + b$, and that is not what you probably wanted. So, all variables in the macro definition, and also macro itself, should be parenthesized:

[21] LP mean Long Pointer, i.e., pointer require 64 bits for storage

```
#define square(x)      ((x)*(x))
```

Example from the minmax.h file from MinGW:

```
#define max(a,b) (((a) > (b)) ? (a) : (b))
...
#define min(a,b) (((a) < (b)) ? (a) : (b))
```

#2

If you define a constant somewhere:

```
#define N 1234
```

... and then redefine it somewhere, compiler will be silent and that may lead to hard-to-find bug.
So that is why it is advisable to define constants as a global variables with a *const* modifier.

1.6 Compiler warnings

Is it worth to turn on `-Wall` in GCC or `/Wall` in MSVC, in other words, to dump all possible warnings? Yes, it is worth to do it, in order to determine quickly small errors. In GCC it is even possible to turn on `-Werror` or `/WX` in MSVC — then all warning will be treated as errors.

1.6.1 Example #1

```
#include <stdio.h>

int f1(int a, int b, int c)
{
        printf ("(in %s) %d\n", __FUNCTION__, a*b+c);
        // return a*b+c; // OOPS, accidentally I forgot to add this
};

int main()
{
        printf ("(in %s) %d\n", __FUNCTION__, f1(123,456,789));
};
```

The author "forgot" to add *return* in f1() function. Nevertheless, GCC 4.8.1 compiles this silently.

It is because in the both C standard ([6, 6.9.1/12]) and in C++ ([8, 6.6.3/2]) is okay if a function does not return a value when it should.

After running we will see this:

```
(in f1) 56877
(in main) 14
```

Where the 14 number is came from? This is what returns the printf() called from f1(). Returned functions results of integral types are left in the EAX/RAX registers. The value from the EAX/RAX register is taken in the main() function and then passed into the second printf() [22].

If to compile with the `-Wall` option, GCC will tell:

```
1.c: In function 'f1':
1.c:7:1: warning: control reaches end of non-void function [-Wreturn-type]
 };
 ^
1.c: In function 'main':
1.c:12:1: warning: control reaches end of non-void function [-Wreturn-type]
 };
 ^
```

[22] About how results are returned via registers, you may read more here [19]

... but will compile anyway.

MSVC 2010 generates the code running likewise, with warning, though:

```
...\1.c(7) : warning C4716: 'f1' : must return a value
```

As you may see, the error is almost critical, caused by, it can be said, type, but there were no compiler warning, or it was inconspicuous.

1.6.2 Example #2

In the C99 standard, new type *bool* and according to standard, it should be big enough to store at least one bit. It is byte in GCC.

If GCC cannot find declaration of some function we going to use, it considers its returning type as *int* by default and warns about it.

Now let's consider we have two files:

Listing 1.2: file1.c

```
bool f1()
{
        ...

        return cond ? true : false;
};
```

Listing 1.3: file2.c

```
...

if (f1())
        do_something();

...
```

GCC has not information about f1() while compiling file2, so it considers its return type as *int*. GCC knows it should return *bool* while compiling file1.c, but byte is enough. Variables of integral types are returned via EAX or RAX registers of x86-processors [23], so GCC generates a code which can set only low byte of the register (AL) to 1 or 0 and do not touch the rest register part, so there might be random noise left from an other code execution. So, the generated code of f1() may return false by writing 0 into the lowest byte of register EAX/RAX, while other bits will contain noise. From the point of view of file2.c where returning type of f1() is considered to be *int*, the returning value may look like: `0x??????00`, where *?* — random bits. So even if when f1() is returning false, if() condition may be triggered almost always.

This notes author once spent several hours for bug-hunting of such error, and he had to dive into the debugger and assembly listings.

A variant of this bug:

Listing 1.4: file1.c

```
uint64_t f1()
{
        return some_large_number;
};
```

Listing 1.5: file2.c

```
...

uint64_t tmp=f1();

...
```

If the compiler will treat return value type of f1() as int, the 64-bit value will be "clipped" to 32-bit (because, supposedly for better compatibility, int is still a 32-bit type in 64-bit environment).

[23] read more here [19, 1.6] on how integral type variables are returned from functions

1.7 Threads

In the C++11 standard, a new *thread_local* modifier was added, showing that each thread will have its own version of the variable, it can be initialized, and it is located in the TLS[24] :

Listing 1.6: C++11

```
#include <iostream>
#include <thread>

thread_local int tmp=3;

int main()
{
        std::cout << tmp << std::endl;
};
```

[25]

In the resulting executable file, the *tmp* variable will be stored in the TLS.

It is useful for storing global variables like *errno*, which cannot be one single variable for all threads.

1.8 main() function

Standard declaration:

```
int main(int argc, char* argv[], char* envp[])
```

argc will be 1 if no arguments present, 2 — if one argument, 3 — if two, etc.

- argv[0] — current running program name.
- argv[1] — first argument.
- argv[2] — second argument.
- etc.

argv can be enumerated in loop. For example, the program may take a file list in command line (like UNIX *cat* utility does, etc). Dashed options may be supplied in order to distinguish them from file names.

Both *envp[]* and *argc/argv[]* can be omitted in the main() function argument list, and it is correct. Read more here on why it is correct: [19, 1.2.1].

Return clause can be omitted in functions as of C99 (1.6.1) (then the *main()* function will return 0 [26]).

in CRT[27] the return value of *main()* function is eventually passed to the *exit()* function or *ExitProcess()* in win32. It is usually a return error code which may be checked in command shells, etc. 0 is usually means success, but of course, it is up to author to define (or redefine) his own return codes.

1.9 The difference between stdout/cout and stderr/cerr

stdout is what is dumped to the console with the help of function *printf()* or *cout* in C++. *stdout* is buffered output, so a user, usually not aware of this, sees output by portions. Sometimes, the program output something using *printf()* or *cout* and then crashes. If something goes to the buffer, but buffer did not have time to "flush" into the console, a user will not see anything. This is sometimes unconvenient. Thus, for dumping more important information, including debugging, it is more convenient to use *stderr* or *cerr*.

stderr is not bufferized output, so, anything comes in this stream with the help of `fprintf(stderr,...)` or *cerr*, appearing in the console instantly.

It is also worth noting that because of buffer absence, output to the *stderr* is slower.

In order to redirect *stderr* to the other file while process running, this can be specified in the command line (Windows/UNIX):

```
process 2> debug.txt
```

... this will redirect *stderr* into the file specified (because number of the error output stream is 2).

[24] Thread Local Storage
[25] Compiled in GCC 4.8.1, but not in MSVC 2012
[26] this rule exception is present only for *main()*
[27] C runtime library

1.10 Outdated features

1.10.1 register

This keyword was used in past to mark a variables which compiler should (if possible) to allocate in CPU registers for the faster access to them.

```
void f()
{
        int a, b;
        register int x, y;
        ...
}
```

Modern compilers are advanced enough to make such decisions on their own, so this keyword is outdated. However, it might be useful while reading ancient source code for quickly spotting busiest variables.

Chapter 2

C

2.1 Memory in C

Probably, there are two most common memory types available for a programmer in C.

- Memory space in the local stack. It is local variables, a memory allocated with the help of alloca(). It is usually a memory very fast to allocate.
- Heap. It is what is allocated with the help of malloc().

2.1.1 Local stack

If you declare something like `char a[1024]`, there is no memory allocation happens, it is just stack pointer moving back for 1024 bytes [19, 1.2.3]. This is a very fast operation.

One not need to free that memory, it is happen automatically at the function end, with the stack pointer restoring.

As the flip side, one need to know exactly how much space to allocate, and also, the block cannot be shrinked or expanded, freed and reallocated again.

Local variables allocated in the local stack by simple shifting stack pointer back [**?**, 1.2.1]REBook]. During that, nothing else is happen, new variables will contain the values which were at the place in stack, most likely, what was left there from previous functions execution.

2.1.2 alloca()

alloca() function likewise allocates a memory block in the local stack, shifting stack pointer [19, 1.2.4]. The memory block will be freed at the function finish automatically.

In the C99(2.5.10) standard, it is necessary to use alloca(), one can write just:

```
void f(size_t s, ...)
{
        char a[s];
};
```

This is called variable length array.

Internally it works just as alloca() however.

Criticism: Linus Torvalds against usage of alloca() [18].

2.1.3 Allocating memory in heap

The heap is an area of memory allocated by OS to the process, where it can divide it within its sole discretion. After terminating of the process (including process crash), the heap is annuled automatically and OS will not need to free all allocated blocks one by one.

There are standard C functions to work with the heap: malloc(), calloc(), realloc(), free(), and new/delete in C++.

Apparently, heap manager must use a lot of interconnected structures in order to preserve information about allocated blocks. So that's why quite tangible overhead is present. You can allocate memory block of size 8 bytes, but at least more 8 bytes[1] will be used for preserving information about allocated block [2]. In 64-bit OS pointers requiring twices as much space,

[1] MSVC, 32-bit Windows, almost the same in Linux

[2] It is also called "metadata", i.e., data about data

so information about each block will require at least 16 bytes. In the light of this, in order to effectively use as much memory as possible, the blocks should be as large as possible, or, the orgranization of data must be different.

Heap using requires a programmer's discipline, it is easy to make a lot of mistakes without one. Probably because of this, it is widely considered that PL with RAII[3] like C++ or PL with garbage collector (Python, Ruby) are easier.

One of the common mistakes: memory leaks

Memory was allocated, but we forgot to free it via free(). This problem is easily solved by thunk functions on top of malloc()/free(). Let this thunk to keep a records about blocks allocated, and also, where and when (and for what) each block was allocated.

I made this in my octothorpe library [4]. DMALLOC macro calls dmalloc() function passing it the file name, name of the calling function, line number and comment (block name). At the end of program, we call `dump_unfreed_blocks()` and it will dump the list of blocks we forgot to free:

```
seq_n:2, size: 124, filename: dmalloc_test.c:31, func: main, struct: block124
seq_n:3, size: 12, filename: dmalloc_test.c:33, func: main, struct: block12
seq_n:4, size: 555, filename: dmalloc_test.c:35, func: main, struct: block555
```

Each block also has a number. This is helpful because one can set a breakpoint by a block number and debugger will trigger at the moment the block is being allocated, and you can see, where and under what conditions it is occurring.

It is boring to write a comment for each block allocated, but very useful. Then it is easy to see, what was memory allocated for. I first saw this idea in the Oracle RDBMS. Aside from that, it also keeps statistics of block types, how many memory was allocated for each, and it is easy to see it:

```
SQL> select * from v$sgastat;

POOL         NAME                            BYTES     CON_ID
------------ -------------------------- ---------- ----------
shared pool  AQ Slave list                     1224          1
shared pool  KQR L PO                        653312          2
shared pool  KQR X SO                        635808          2
shared pool  RULEC                            20688          1
shared pool  KQR M SO                          7168          2
shared pool  work area table entry            12240          2
shared pool  kglsim object batch               3864          2
large pool   PX msg pool                     860160          1
large pool   free memory                   30523392          0
large pool   SWRF Metric CHBs                1802240          2
large pool   SWRF Metric Eidbuf               368640          2
```

The same thing present in the Windows kernel, it is called *tagging*.

When one allocates memory in the kernel or driver, a 32-bit tag may be set (usually, it is a four-letter abbreviation, indicating Windows subsystem). Then it is possible to see statistics in a debugger, how much memory is allocated what for:

```
kd> !poolused 4
   Sorting by  Paged Pool Consumed

  Pool Used:
            NonPaged            Paged
 Tag    Allocs     Used    Allocs     Used
 CM25        0        0       935  4124672      Internal Configuration manager allocations ,
    Binary: nt!cm
 Gh05        0        0       268  3291016      GDITAG_HMGR_SPRITE_TYPE , Binary: win32k.sys
 MmSt        0        0      2119  2936752      Mm section object prototype ptes , Binary: nt!mm
 CM35        0        0        91  2150400      Internal Configuration manager allocations ,
    Binary: nt!cm
 vmfb        0        0        13  2148752      UNKNOWN pooltag 'vmfb', please update pooltag.txt
 Ntff        5     1040      1287  1070784      FCB_DATA , Binary: ntfs.sys
 ArbA        0        0       108   442368      ARBITER_ALLOCATION_STATE_TAG , Binary: nt!arb
 NtfF        0        0       457   431408      FCB_INDEX , Binary: ntfs.sys
```

[3] Resource Acquisition Is Initialization

[4] https://github.com/dennis714/octothorpe/blob/master/dmalloc.c

```
CM16        0        0        62   331776      Internal Configuration manager allocations ,
   Binary: nt!cm
IoNm        0        0      2022   267288      Io parsing names , Binary: nt!io
Ttfd        0        0       159   253976      TrueType Font driver
Ifs         0        0         4   249968      Default file system allocations (user's of ntifs.
   h)
CM29        0        0        26   212992      Internal Configuration manager allocations ,
   Binary: nt!cm
```

Of course, one may argue the heap manager will require much more space about allocated blocks, including their names or tags. And it is slowing down the program much more. That is for sure. Then we may use it only in debug builds, and in the release-builds DMALLOC() will be simple *empty* thunk-function for malloc(). It is also turned off by default in Windows and it must be turned on with the help of GFlags utility [5] Aside from that, something similar present in MSVC [6].

One of the common mistakes: heap corruption

It is easy to allocate a memory for 4 bytes, but write there a fifth by accident. Most likely, it will not come out instantly, but in fact, it is a very dangerous time bomb, dangerous because it is hard-to-find bug. The byte next after block you allocated, most likely, is not used at all, but there may begin a heap manager structure, keeping the information about some other allocated block, or maybe even that block. If some of these structures get corrupted or rewriten intentionally, consequent malloc() or free() calls will not work properly. Sometimes it is manifested in errors like (in Windows):

```
HEAP[Application.exe]: HEAP: Free Heap block 211a10 modified at 211af8 after it was freed
```

Such errors are exploited by exploit authors: if you know you can alter heap manager structures in a way you need, you may achieve some specific program behaviour you need (this is called heap overflow [7]).

Widely used protection from such errors: just to write "guard" values (e.g. of 32-bit size) at the both sides of the block. For example, I did it in DMALLOC. At each free() call, integrity of both guards are checked (these may be a fixed values like 0x12345678), and if something or someone wrote to it, that fact can be reported instantly.

One of the common mistakes: not checking malloc() result

If malloc() finishes successfully, it returns a pointer to the newly allocated block that can be used, or NULL in case of memory shortage. Of course, in our time of cheap memory, this is rare problem, nevertheless, if one uses it a lot, one should consider it. It is not handy to check the returned pointer after each malloc() call, so there are a popular technique to write own thunk functions named xmalloc(), xrealloc() calling malloc()/realloc(), which checks returning result and exiting in case of error.

It is interesting to note how xmalloc() behaves in git:

```
void *xmalloc(size_t size)
{
        void *ret;

        memory_limit_check(size);
        ret = malloc(size);
        if (!ret && !size)
                ret = malloc(1);
        if (!ret) {
                try_to_free_routine(size);
                ret = malloc(size);
                if (!ret && !size)
                        ret = malloc(1);
                if (!ret)
                        die("Out of memory, malloc failed (tried to allocate %lu bytes)",
                            (unsigned long)size);
        }
#ifdef XMALLOC_POISON
        memset(ret, 0xA5, size);
#endif
        return ret;
}
```

[5] http://msdn.microsoft.com/en-us/library/windows/hardware/ff549557(v=vs.85).aspx
[6] read more about the _CrtSetDbgFlag and _CrtDumpMemoryLeaks functions
[7] https://en.wikipedia.org/wiki/Heap_overflow

[8]

If malloc() is not successful, it tries to free some already allocated (but not very needed) blocks with the help of try_to_free_routine(), and then to call malloc() again.

Aside from that, if XMALLOC_POISON macro is defined, all bytes in the block allocated is filled with 0xA5.

This may help to see, visually, when you use some value from the block before its initialization.

The value of `0xA5A5A5A5` will easily be spotted in the debugger, or, in some place in dump where it will be printed in hexadecimal form. There are the constant for the same purpose in MSVC: `0xbaadf00d`.

Even more than that: after call of free(), freed block may be marked by some other constant, in order to spot visually if someone attempting to use some data from the block after it has been freed.

Some constants from Microsoft:

```
* 0xABABABAB : Used by Microsoft's HeapAlloc() to mark "no man's land" guard bytes after
    allocated heap memory
* 0xABADCAFE : A startup to this value to initialize all free memory to catch errant pointers
* 0xBAADF00D : Used by Microsoft's LocalAlloc(LMEM_FIXED) to mark uninitialised allocated heap
    memory
* 0xCCCCCCCC : Used by Microsoft's C++ debugging runtime library to mark uninitialised stack
    memory
* 0xCDCDCDCD : Used by Microsoft's C++ debugging runtime library to mark uninitialised heap
    memory
* 0xFDFDFDFD : Used by Microsoft's C++ debugging heap to mark "no man's land" guard bytes before
    and after allocated heap memory
* 0xFEEEFEEE : Used by Microsoft's HeapFree() to mark freed heap memory
```

[9]

Other common mistakes

If you do not include header file stdlib.h, GCC treat the returning value of malloc() as *int* and warns about types.

On the other hand, in C++, malloc() result should be casted anyway to right type:

```
int *a=(int*) malloc(...);
```

Another mistake may cost your nerves is to allocate the same block of memory at one place more than one time (previous calls are "hiding" from the sight).

Other bug hunting techniques

However, you may be in the situation, when there are bugs in program, but you are not able to recompile it for some reason. As an example, valgrind[10] may help then.

2.1.4 Local stack or heap?

Of course, allocation in the local stack is much faster.

For example: in tracer [11] i have a disassembler[12] and also x86 CPU emulator [13]. When I was writing the disassembler in C (I did it after a long period of programming in higher level PL — Python), I thought, it is a good idea for disassembler to allocate the memory for the structure it returns, fill it and return a pointer to it, or NULL in case of disassembling error.

Æsthetically it looks good, in style of high-level PL, aside from that, such code is easier to read. However, disassembler and x86 CPU emulator works in loop, a huge number of iterations per second and efficiency is crucial. So the main loop I wrote in that way:

```
while(true)
{
        struct disassembled_instruction DA;

        bool DA_success=disassemble(&DA...);
        if (DA_success==false)
                break;
```

[8] https://github.com/git/git/blob/master/wrapper.c
[9] https://en.wikipedia.org/wiki/Magic_number_(programming)
[10] http://valgrind.org/
[11] http://yurichev.com/tracer-en.html
[12] https://github.com/dennis714/x86_disasm
[13] https://github.com/dennis714/bolt/blob/master/X86_emu.c

```
        bool emulate_success=try_to_emulate(&dDA);
        if (emulate_success==false)
                break;

};
```

There are no costs for allocating disassembler structures at all. Otherwise, we need to call malloc()/free() at each loop iteration, each of which will also work with heap data structures, etc.

As we know, x86-instructions may have up to 3 operands, so, in my structure, aside from instruction code, there are also information about 3 operands. Of course, I could do it like:

```
struct disassembled_instruction
{
        int instruction_code;
        struct operand *op1;
        struct operand *op2;
        struct operand *op3;
};
```

... and let it be NULL there in case of absence of some operand. Nevertheless, it is still heap memory allocations.

So I did it like:

```
struct disassembled_instruction
{
        int instruction_code;
        int operands_total;
        struct operand op[3];
};
```

Such structure requires more memory. Aside from that, 3-operand instructions are rare in x86-code, but third operand is stored here always. However, there are no extra manipulations with memory.

But if one likes to save space by not storing the third operator, it may be not stored at all: it is easy to calculate structure size without one operatnd: `sizeof(disassembled_instruction) - sizeof(struct operand)` and copy it to the some place where it must be stored. Because no one prohibits to use (and store) not the whole structure, but only its part. Besides, the functions which work with the structure, may not touch third operand at all, and that will work correctly.

Even more than that: I made my disassembler intentionally in that way that it can take not initialized structure and may work even if there is still some information left from the previous calls.

Maybe it is overkill, but you got the idea.

Thus, if you allocate small structures of known size and if speed is crucial, you may consider allocating them in the local stack.

2.2 Strings in C

The reason why C string format is as it is (zero-terminating) is apparently historical. In [15] we can read:

> A minor difference was that the unit of I/O was the word, not the byte, because the PDP-7 was a word-addressed machine. In practice this meant merely that all programs dealing with character streams ignored null characters, because null was used to pad a file to an even number of characters.

There are no features in C to handle strings like those present in higher level PLs like concatenation.

People often complain about awkward string concatenation (i.e., glueling together). Also irritating sprintf(), for which is hard to predict how much space will need.

String copying with strcpy() is not easy as well — one needs to think ahead how many bytes must be allocated for buffer. Aside from that, awkward C strings is the source of huge number of vulnerabilities related to buffer overflow [19, 1.14.2].

In the first place, we should ask ourselves, which string operations we need. Concatenation (glueling) is needed for 1) output messages to log; 2) construction of strings and then to pass (or write) them to some place.

For 1) it is possible to use streams — without string construction just to output it by portions, e.g.:

```
printf ("Date: ");
dump_date(stdout, date);
printf (" a=");
dump_a(stdout, a);
printf ("\n");
```

This is what *ostream* in C++ is intended for:

```
cout << "Date: " << Date_ToString(date) << " a=" << a_ToString(a) << "\n";
```

It is faster, and requries less memory for string construction.
By the way, it is a mistake to write like:

```
cout << "Date: " + Date_ToString(date) + " a=" + a_ToString(a) + "\n";
```

At an easy pace, it is good enough to write messages to log, however, if there are a lot of such messages, there may be string concatenation overhead.

Anyway, sometimes strings must be constructed.
There are some libraries for this. For example, in Glib[14] there exist gstring.h[15]/ gstring.c[16].
In the git source code we may find strbuf.h[17]/ strbuf.c[18]. Strictly speaking, such C-libraries are very similar: they provide a data structure with a string buffer in it, current buffer length and current string in buffer length. With the help of various functions, it is possible to add to buffer other string or characters, which, in turn, will grow or shrink.
In *strbuf.c* from git there is also a function *strbuf_addf()*, working just like *sprintf()*, but adding resulting string into the buffer.
Thus a programmer may get rid of headache related to memory allocation. While using such libraries, buffer overflows are virtually impossibly if not to work with the structures by himself.
The typical sequence of using such libraries looks like:

- Structure strbuf or GString initialization.
- Adding strings and/or characters.
- Now we have constructed string.
- Modifying it if need.
- Using it as usual C-string, writing it to to some file, send it by network, etc.
- Structure deinitialization.

By the way, string construction resembles somehow Buffer[19], ByteBuffer[20] and CharBuffer[21] in Java.

2.2.1 String length storage

String length is always stored — it was done in Pascal PL implementations. Aside from holy wars outcomes between both PL devotees, nevertheless, almost all string libraries keep current string length — just because conveniences outweigh the need of length value recalculation after each modification.
For example, *strlen()* [22] is not needed at all, string length is always known. String concatenation is also much faster, because we do not need to calculate length of the first string. The function of strings comparing may just compare string lengths at the beginning and if they are not equal to each other, return *false* without starting to compare characters in the strings.
In the network libraries of Oracle RDBMS, to the various string functions often passed string with its length, as separate argument [23]. Not very æsthetical, looks redundant, but very useful. For example, we have a function, which needs to know, which string was passed to it:

[14] https://developer.gnome.org/glib/
[15] https://github.com/GNOME/glib/blob/master/glib/gstring.h
[16] https://github.com/GNOME/glib/blob/master/glib/gstring.c
[17] https://github.com/git/git/blob/master/strbuf.h
[18] https://github.com/git/git/blob/master/strbuf.c
[19] http://docs.oracle.com/javase/7/docs/api/java/nio/Buffer.html
[20] http://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html
[21] http://docs.oracle.com/javase/7/docs/api/java/nio/CharBuffer.html
[22] string length calculation
[23] http://blog.yurichev.com/node/64

```
void f(char *color)
{
        if (strcmp (color, "red")==0)
                do_red();
        else if (strcmp (color, "green")==0)
                do_green();
        else if (strcmp (color, "blue")==0)
                do_blue();
        else if (strcmp (color, "orange")==0)
                do_orange();
        else if (strcmp (color, "yellow")==0)
                do_yellow();
        printf ("Unknown color!\n");
};
```

However, if this function have length of the input string, it may be rewritten like:

```
void f(char *color, int color_len)
{
        switch (color_len)
        {
                case 3:
                        if (strcmp (color, "red")==0)
                                do_red();
                        else
                                goto unknown_color;
                        break;
                case 4:
                        if (strcmp (color, "blue")==0)
                                do_blue();
                        else
                                goto unknown_color;
                        break;
                case 5:
                        if (strcmp (color, "green")==0)
                                do_green();
                        else
                                goto unknown_color;
                        break;
                case 6:
                        if (strcmp (color, "orange")==0)
                                do_orange();
                        else if (strcmp (color, "yellow")==0)
                                do_yellow();
                        else
                                goto unknown_color;
                        break;
                default:
                                goto unknown_color;

        };

        return;

unknown_color:
        printf ("Unknown color!\n");
};
```

Æsthetically, the code looks just horrible. Nevertheless, we got rid of a lot of strings comparison calls! Apparently, for those cases when strings must be processed fast, such approach may help.

2.2.2 String returning

If a function must return a string, these options are available:

- 1: Constant string returning, is simplest and fastest.
- 2: String returning via global array of characters. Shortcoming: there is only one array and each subsequent function call overwrites its contents.
- 3: String returning via buffer, pointer to which is passed in the function arguments. Shortcoming: buffer length must be passed as well, and also its length cannot be correctly calculated in before.
- 4: Allocate buffer of a size we need on our own, write string to it, return the pointer to the buffer we allocated. Shortcoming: resources spent on memory allocation.
- 5: Write the string to the `strbuf` we already mentioned or `GString` or any other structure, pointer to which was passed in the arguments.

2.2.3 1: Constant string returning

The first option is very simple. E.g.:

```
const char* get_month_name (int month)
{
        switch (month)
        {
        case 1: return "January";
        case 2: return "February";
        case 3: return "March";
        case 4: return "April";
        case 5: return "May";
        case 6: return "June";
        case 7: return "July";
        case 8: return "August";
        case 9: return "September";
        case 10: return "October";
        case 11: return "November";
        case 12: return "December";
        default: return "Unknown month!";
        };
};
```

Even simpler:

```
const char* month_names[]={
        "January", "February", "March", "April", "May", "June", "July", "August",
        "September", "October", "November", "December" };

const char* get_month_name (int month)
{
        if (month>=1 && month<=12)
                return month_names[month-1];

        return "Unknown month!";
};
```

2.2.4 2: Via global array of characters

That is how `asctime()` does it. Keep in mind that the string should be used before each subsequent call to `asctime()`.

For example, this is correct:

```
printf("date1: %s\n", asctime(&date1));
printf("date2: %s\n", asctime(&date2));
```

This is not:

```
char *date1=asctime(&date1);
char *date2=asctime(&date2);
printf("date1: %s\n", date1);
printf("date2: %s\n", date2);
```

... because `date1` and `date2` pointers will point to one place and `printf()` output will be the same.

In *hex.c* of git[24] we may find this:

```
char *sha1_to_hex(const unsigned char *sha1)
{
        static int bufno;
        static char hexbuffer[4][50];
        static const char hex[] = "0123456789abcdef";
        char *buffer = hexbuffer[3 & ++bufno], *buf = buffer;
        int i;

        for (i = 0; i < 20; i++) {
                unsigned int val = *sha1++;
                *buf++ = hex[val >> 4];
                *buf++ = hex[val & 0xf];
        }
        *buf = '\0';

        return buffer;
}
```

In fact, the string is returned via global variable, `static` declaration makes it visible only from this function. Here is a shortcoming: after call to *sha1_to_hex()* you cannot call it again for the second string result before you use the first somehow, because it will be overwritten. Apparently, in order to solve the problem, there are 4 buffers, and the string is returned each time in the next one. It is also worth to notice — it is possible to do such things if you are sure of what you do, the code is on the "dirty hack" level. If you will call this function 5 times and will need to use the first string somehow, this may lead to hard-to-find bug.

You may also notice that *bufno* is not initialized, because only 2 lower bits are used, aside from that, it is not important at all, which value it will hold at the program start.

2.2.5 Standard string C functions

Some functions like getcwd() not only fill the buffer, but also returns a pointer to it. It is made for the situations, where it is more compact to write something like:

```
char buf[256];
do_something (getcwd (buf, sizeof(buf)));
```

... instead of:

```
char buf[256];
getcwd (buf, sizeof(buf))
do_something (buf);
```

strstr() and memmem()

strstr() is intended for searching for a substring in another string, or to get to know, are there substring present in it anyway.

memmem() can be used with the same intentions, but for searching in the buffer which may contain zeroes, ot in the part of a string.

strchr() and memchr()

strchr() is used for searching for character in a string or to get to know if there are such characters present.

memchr() can be used with the same intentions, but for searching in the part of a string.

[24] https://github.com/git/git/blob/master/hex.c

atoi(), atof(), strtod(), strtof()

atoi()/atof() cannot signal an error, but strtod()/strtof() while doing the same thing — can signal.

scanf(), fscanf(), sscanf()

A well-known holy-war, is text files are better than binary files or otherwise. It is easier and faster to process binary files, however, text files are easier to view and edit in any text editor, beside, UNIX has a lot of utilities for text and strings processing. But text files must be parsed.

scanf() function [6, 7.19.6/2] of course, does not support regular expressions, however, some simple sequences can be parsed by it.

Example #1 The `/proc/meminfo` file generated by Linux kernel, beginning as:

```
MemTotal:        1026268 kB
MemFree:          119324 kB
Buffers:          170796 kB
Cached:           263736 kB
SwapCached:        11428 kB
...
```

Let's consider, we need to get first and third numbers, ignoring second and rest. That is how it can be done:

```
void read_proc_meminfo()
{
        FILE *f=fopen("/proc/meminfo", "r");
        assert(f);
        unsigned result1, result2;
        if (fscanf (f, "MemTotal:\t%d kB\n"
                       "MemFree:\t%*d kB\n"
                       "Buffers:\t%d kB\n",
                       &result1, &result2)==2)
                printf ("results: %d %d\n", result1, result2);
        fclose(f);
};
```

The format string is defined in three lines, it is one in fact: (1.2.2). N.B. The newline is defined as `\n`.

`*` in the scanf-string modifier pointing out that the number will be read, but will not be stored. Thus, the field is being ignored. scanf()-functions are returning not a number of fields read (3 will be here), but number of fields stored (2 will be here).

Example #2 There is a text file containing key-value pairs in each string:

```
some_param1=some_value
some_param2=Lazy fox etc etc.
param3=Lorem Ipsum etc etc.
space here=value containing space
too long param, we should fail here=value
```

We should just read two fields:

```
int main(int argc, char *argv[])
{
        assert(argc==2);
        assert(argv[1]);
        FILE *f=fopen (argv[1], "r");
        assert(f);
        int line=1;
        do
        {
                char param[16];
                char value[60];
                if (fscanf (f, "%16[^=]=%60[^\n]\n", param, value)==2)
```

```
                {
                        printf ("param=%s\n", param);
                        printf ("value=%s\n", value);
                }
                else
                {
                        printf ("error at line %d\n", line);
                        return 0;
                };
                line++;
        } while (!feof(f) && !ferror(f));
};
```

%16[^=] — is somewhat looks like regular expression. Meaning, to read any 16 characters, except "equal" (=) sign. Then we point to scanf() that there must be this sign (=). Then let him to read any 60 characters. We read newline character at the end.

This works, and field lengths are limited to 16 and 60 characters. That is why error predictabily occuring on the fifth string, because it has larger length of parameter (first field).

Thus it is possible to parse simple file formats, even CSV[25].

However, it should be noted that scanf()-functions are not able to read empty string where %s modifier is defined. Thus it is not possible to parse a key-value file with absent keys or values.

Caveat #1 scanf() treat %d modifier in the format string as 32-bit `int` on both x86 and x64 CPUs.

It is a common mistake to write:

```
char a[10];

scanf ("%d %d %d %d", &a[0], &a[1], &a[2], &[3]);
```

Characters (or bytes) are placed adjacently to each other. When scanf() will process first value, it will treat it as 32-bit `int` and overwrite other 3 located near. And so on.

strspn(), strcspn()

`strspn()` is often used to get to be sure that a string has only characters from the list we defined:

```
if (strspn(s, "1234567890") == strlen(s)) ... OK
...
if (strspn(IPv4, "1234567890.") == strlen(IPv4)) ... OK
...
if (strspn(IPv6, "0123456789AaBbCcDdEeFf:.") == strlen(IPv6)) ... OK
```

Or to skip the beginning of a string:

```
const char *whitespaces = " \n\r\t";
*buf += strspn(*buf, whitespaces); // skip whitespaces at start
```

`strcspn()` is inverse function, it can be used for skipping all symbols at the string beginning, which are not defined in a set:

```
s += strcspn(s, whitespaces); // first, skip anything till whitespaces
s += strspn(s, whitespaces); // then skip whitespaces
// here 's' is pointing to the part of string after whitespaces
```

strtok() and strpbrk()

Both functions are used for delimiting string into substrings, divided by special characters [26]. However strtok() modifies source string (and thus resulting substrings can be used as separated C-strings), but strpbrk() is not, it is only returning a pointer to the next substring.

[25] Comma-separated values
[26] delimiter

2.2.6 Unicode

Unicode is important these days. Most popular approaches are:

- UTF-8 Popular in Unices. Significant advantage: it is still possible to use many (but not limited to) standard functions for strings processing.
- UTF-16 Used in Windows API.

UTF-16

For each character a 16-bit type is assigned: *wchar_t*.

For such typed string definition, *L* macro is used::

```
L"hello world"
```

There is a special class of "twin" functions with the "w" in name, intended for work with wchar_t instead of char: fwprintf(), wcscmp(), wcslen(), iswalpha().

Windows There is `tchar` type in the Windows API which helps us to write a program in two builds: with Unicode and without, depending on `UNICODE` preprocessor variable definition, it will be `char` or `wchar_t` [27]. `_T(...)` macro is also intended for this:

```
_T("hello world")
```

Depending on `UNICODE` preprocessor macro definition, it will be `char` or `wchar_t`.

In the `tchar.h` header file, there are a lot of functions, changing its behaviour depending on this variable.

2.2.7 Lists of strings

The simplest list of strings is just a strings set ending with the zero. For example, in Windows API, in the Common Dialogs library, thus [28] a list of available file extensions for dialog box are passed:

```
// Initialize OPENFILENAME
ZeroMemory(&ofn, sizeof(ofn));
...
ofn.lpstrFilter = "All\0*.*\0Text\0*.TXT\0";
...

// Display the Open dialog box.

if (GetOpenFileName(&ofn)==TRUE)
        ...
```

2.3 Your own data structures in C

2.3.1 Lists in C

Lists are linked set of elements. Singly-linked list — it is when each element has pointer to the next one. Doubly-linked list — is when each element has pointers to the both previous element and the next one.

In comparison with arrays, one significant advantage is ease of new element adding at the random place. As disadvantages: list supporting data structures consumes some memory overhead, and also it is not possible to index a list as an array.

Singly-linked list

Simplest to implement. In the structure intended for linking into a list, it is enough just to add somewhere a link to the next element, usually this field called `next`:

[27] Simultenous builds with Unicode and without were popular in the time of popularity of both Windows NT/2000/XP and Windows 95/98/ME lines. Unicode support in the second was not very good

[28] http://msdn.microsoft.com/en-us/library/windows/desktop/ms646829(v=vs.85).aspx

```
struct some_object
{
        ...
        ...
        struct some_object* next;
};
```

NULL in the next meaning that the element is last in the list.

Elements enumerating in this list is straightforward:

```
for (struct some_object *i=list; i!=NULL; i=i->next)
        ...
```

One need first to find the last element:

```
for (struct some_object *i=list; i->next!=NULL; i=i->next);
struct some_object *last_element=i;
```

... and then, after creating new structure, store the pointer to it in next:

```
struct some_object *new_object=calloc(1, sizeof(struct some_object));
// populate new_object with data
last_element->next=new_object;
```

calloc() is different from the malloc() in the sense that all allocated space will be cleared and consequently, there will be NULL in the next field [29].

Searching for the needed element is just enumerating all elements in the list with comparing them with the sought-for element until it is found.

Element deletion: find the previous element and the next, set the next pointer in the previous element to the next one, then free memory block allocated for the current element.

The very first list element is also called "list head". The very first element structure can be declared as a local or global variable. But it will be harder to delete first element. On the other hand, it is possible to declare the pointer to the first list element, then it will be easier to assign other element to this pointer (which will be first).

Doubly-linked list

Almost the same, but, aside from the pointer to a next element, also pointer to a previous element is stored. If the element is first, the pointer to the previous element may be NULL, or it may point to itself (whatever you like).

When working with doubly-linked list, it is easier to find previous elements, e.g., in case of element deletion. It is easier to enumerate elements backwards from the end of the list. But the memory overhead is slightly larger.

Often, doubly-linked list is also circular, i.e., the first and the last elements are pointing to each other. For example, that is how it is done in std::list in C++ STL [19, 2.4.2]. This simplifies searching of the last element (one does not need to enumerate all elements).

Windows API

Here, and also in a lot of places in the Windows kernel, two primitive data structures are used:

```
typedef struct _LIST_ENTRY {
   struct _LIST_ENTRY *Flink;
   struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY, *RESTRICTED_POINTER PRLIST_ENTRY;

typedef struct _SINGLE_LIST_ENTRY {
    struct _SINGLE_LIST_ENTRY *Next;
} SINGLE_LIST_ENTRY, *PSINGLE_LIST_ENTRY;
```

These structures are not intended for independent use, but rather they are intended for embedding into another structures. For example, we need to unite a color-describing structure into a list:

[29] More about structures "initialization", read here: (2.4.1).

```
struct color
{
        int R;
        int G;
        int B;
        LIST_ENTRY list;
};
```

Now we have a pointers to the both next and previous elements. There is a small API present in Windows API using these structures [30].

Linux

Doubly-linked list routines in Linux kernel are declared in the file /include/linux/list.h [31].

It is heavily used there, in the kernel version 3.12 there are at least 2900 references to "struct list_head".

Glib

One might ask, is not it possible to declare a perticular structure for the list element, and not to embed it to own structures? Yes, for example, that is how it is done in glist.h [32] in Glib:

```
struct _GList
{
  gpointer data;
  GList *next;
  GList *prev;
};
```

`data` may point to any object you like, to any existing structure in which you want not to change anything, this is also called "opaque pointer". Of course, æsthetically it is better. But one should remember that there will be two allocated memory block for each element of list + memory overhead for supporting allocated blocks in heap(2.1.3).

Thus, this approach is acceptable if memory footprint is not important.

2.3.2 Binary trees in C

Binary trees — are one of the most important structures in computer science. Most often these are used for "key-values" pairs storage. This is what implemented in std::map in C++ STL[33].

Simply speaking, in comparison with lists, trees offer much faster selection. On the other hand, element insertion may be slower.

There are no C standard functions for working with trees, but some functions are present in POSIX[34] (tsearch(), twalk(), tfind(), tdelete()) [35].

This family of functions are used actively in the Bash 4.2, BIND 9.9.1, GCC — it can be seen there how it can be used.

The Glib also has the tree functions declared in the gtree.h [36].

The set (std::set in C++ STL) can be implemented as binary trees as well, one may just choose not to store the value and store the key only.

2.3.3 One more thing

Data structures related to collections may also contain pointers to the functions working with elements, like comparison functions, copying, etc.

For example in GTree in Glib:

[30] http://msdn.microsoft.com/en-us/library/windows/hardware/ff563802(v=vs.85).aspx
[31] http://lxr.free-electrons.com/source/include/linux/list.h
[32] https://github.com/GNOME/glib/blob/master/glib/glist.h https://developer.gnome.org/glib/2.37/glib-Doubly-Linked-Lists.html
[33] Standard Template Library
[34] Portable Operating System Interface
[35] http://pubs.opengroup.org/onlinepubs/009696799/functions/tsearch.html
[36] https://github.com/GNOME/glib/blob/master/glib/gtree.h

Listing 2.1: gtree.c

```
struct _GTree
{
  GTreeNode        *root;
  GCompareDataFunc  key_compare;
  GDestroyNotify    key_destroy_func;
  GDestroyNotify    value_destroy_func;
  gpointer          key_compare_data;
  guint             nnodes;
  gint              ref_count;
};
```

By setting the functions for key/value comparison and also deallocator function (in `g_tree_new_full()`), tree functions in Glib will be able to compare two trees or to free a tree on its own.

2.4 Object-oriented programming in C

Of course, there is no OOP support in C, it is present in C++, nevertheless, it is possible to program in OOP style in "pure" C.

OOP, in short, is a separation to object and methods. In C, structures can easily be represented as objects, and usual functions — as methods.

2.4.1 Structures initialization

C++ has class constructors. If one needs to initialize structure in some special way, one would write a special function for it in C as well. But if it simple structure, it is possible to initialize it with calloc() [37] or bzero()(2.5.3).

All int-variables are set to zeroes. Zero value bool in C99(2.5.10) and C++ is false, same as BOOL in Windows API. All pointers are set to NULL. And even floating point 0.0 in IEEE 754 format is zero bits in all positions.

If a structure has pointers to other structures, NULL can mean "object absence".

Among other things, not initialized global variables are also zeroed [6, 6.7.8.10].

Initialization is an important thing. It is very hard to catch a bug related to non initialized variables accesses. The compiler will not warn you if you use a structure field without initialization.

2.4.2 Structures deinitialization

If a structure has pointers to other structures, they are also must be freed. In simple case, it is just a call to free(). By the way, that is why NULL is valid argument for free(), it allows to write `free(s->field)` instead of `if (s->field) free(s->field)`, that is shorter.

2.4.3 Structures copying

If a structure is simple, it is possible to copy it with a call to memcpy()(2.5.2). If you are copying structures having pointers to other structures in this manner, it's called "shallow copy"[38]. And in opposite, *deep copy* — is copying a structure with all structures connected to it (slower operation).

That is why it may be more convenient to store a string in a structure as a fixed-size array of characters. For example, a lot of such cases in the Windows API. Such a structure is easier to copy, it requires smaller memory overhead in the heap. On the other hand, we should accept string length fixedness.

Aside from that, a structure can be copied just as: `s1=s2` — the code generated will copy each structure field. Perhaps it is easier to read than a call to memcpy() at the same place.

2.4.4 Encapsulation

C++ offers encapsulation (information hiding). For example, you cannot write a program which modifies a protected class field, this is a compile-stage protection [19, 1.7.3].

There is no such thing in C, it requires more discipline.

However, it is possible to "protect" a structure from "prying eyes". For example, Glib has a library intended for working with trees. In the header file gtree.h[39] there is no declaration of the structure (it is present only in the gtree.c [40]), there are

[37] It is the same as the malloc() + allocated memory filling with zeroes

[38] https://en.wikipedia.org/wiki/Object_copy

[39] https://github.com/GNOME/glib/blob/master/glib/gtree.h

[40] https://github.com/GNOME/glib/blob/master/glib/gtree.c

only forward declaration(1.2.1). Thus Glib developers may have a hope that GTree users will not try to use specific fields in the structure directly.

The technique does have its flip side: there may be tiny one-line functions like "return string length" in strbuf(2.2), e.g.:

```
typedef struct _strbuf
{
    char *buf;
    unsigned strlen;
    unsigned buflen;
} strbuf;

unsigned strbuf_get_len(strbuf *s)
{
        return s->strlen;
};
```

If the compiler during the compiling stage has access to a structure declaration and the function body, instead of call to strbuf_get_len() it may make this function as inline, i.e., insert its body right at the place and save some resources on call to another function. But if this information is not available to the compiler, it will leave call to strbuf_get_len() as is.

The same thing is applies to the `buf` field in the `strbuf` structure. Compiler may generate much more effective code if the generated machine code will access structure fields directly without calling surrogate functions-"methods".

2.5 C standard library

2.5.1 assert

This macro is commonly used for validating [41] of input values. For eample, if you have a function working with data, you probably may want to add that code at the beginning: *assert (month>=1 && month<=12)*.

Here is what one should remember: standard assert() macro is available only in debug builds. In a release build, where NDEBUG is defined, all statements are "disappearing". That is why it is not correct to write *assert(f=malloc(...))*. However, you may want to write something like *assert(object->get_something()==123)*.

Error messages also can be embedded in an *assert* statements: you will see it if expression will not be true. For example, in the LLVM[42] source code we may find this:

```
assert(Index < Length && "Invalid index!");
...
assert(i + Count <= M && "Invalid source range");
...
assert(j + Count <= N && "Invalid dest range");
```

Text string has *const char** type and it is never NULL. Thus it is possible to add *... && true* to any expression without changing its sense.

assert() macro can also be used for documenting purposes.

For example:

Listing 2.2: GNU Chess

```
int my_random_int(int n) {

   int r;

   ASSERT(n>0);

   r = int(floor(my_random_double()*double(n)));
   ASSERT(r>=0&&r<n);

   return r;
}
```

By reading the code we can quickly see "legal" values of the n and r variables.

assert-s are also called "active comments" [11].

[41] "invariant" and "sanitization" terms are also used

[42] http://llvm.org/

2.5.2 UNIX time

In UNIX-environment UNIX time representation is very popular. It is just a 32-bit number, counting number of seconds passing from the 1st January 1970.

On a positive side: 1) 32-bit number is easy to store; 2) date difference is calculated easily; 3) it is not possible to encode incorrect date and time, like 32nd January, 29th February of non-leap year, 25 hours 62 minutes.

On a negative side: 1) it is not possible to encode a date before year 1970.

If you are to use UNIX time format in modern time, it is worth to remember that it has "expiration date" in year 2038, that will be a year when a 32-bit number will overflow, i.e., 2^{32} seconds will pass since year 1970. So, 64-bit value should be used instead, i.e., `time64`.

2.5.3 memcpy()

It is hard to memorize arguments order in the functions memcpy(), strcpy() at first. It can be easier to memorize if you visualize "=" ("equal") sign between arguments.

2.5.4 bzero() and memset()

`bzero()` is a function which just fills memory block to zeroes. The `memset()` is often used in the place. But the `memset()` has unpleasant feature, it is easy to reverse the second and the third arguments and the compiler will be silent because the filling byte is specified by *int* type.

Aside from that `bzero()` function name is easier to read.

On the other hand, it is absent in C standard, however, it is present in POSIX.

In the Windows API ZeroMemory() [43] exists for the same purpose.

2.5.5 printf()

Limit output string by size:

```
printf ("%.*s", size, buf);
```

Sometimes one need to print identation of N spaces or tabulation symbols before each printing message:

```
const char* ALOT_OF_SPACES="                                                    ";
const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t\t";

printf ("%.*smessage\n", current_nested_level, ALOT_OF_SPACES);
...
printf ("%.*smessage\n", current_nested_level, ALOT_OF_TABS);
```

size_t, ssize_t, DWORD

size_t in MSVC: `"%Iu"`, ssize_t: `"%Id"`.

size_t in GCC: `"%zu"`, and `"%zd"` for ssize_t.

Printing DWORD in MinGW: `"%lu"` or `"%lx"`.

Caveat #1

This code:

```
char c=0x80;
printf ("%02X", c);
```

... dumps FFFFFF80 because *char* is signed by default (as in GCC and in MSVC). While passing it as an argument to *printf()* it's sign-extended to *int* and we see what we see. This is correct:

```
char c=0x80;
printf ("%02X", (unsigned char)c);
```

[43] http://msdn.microsoft.com/en-us/library/windows/desktop/aa366920(v=vs.85).aspx

Your own printf() format-string modifiers

It is often irritating when it is logical to pass to printf(), let's say, a structure describing complex number, or a color encoded as 3 int numbers as a single entity.

In C++ this problem is usually solved by definition `operator«` in `ostream` for the own type, or by a method definition named `ToString()` (3.4).

In printk() (printf-like function in Linux kernel) there exists additional modifiers [44], like `%pM` (Mac-address), `%pI4` (IPv4-address), `%pUb` (UUID[45]/GUID[46]).

In GNU Multiple Precision Arithmetic Library there are gmp_printf() [47] function having non-standard modifiers for BigInt-numbers outputting.

In the Plan9 OS, and in Go compiler source code, we may find fmtinstall() function for a new printf-string modifier definition, e.g.:

Listing 2.3: go\src\cmd\5c\list.c

```
void
listinit(void)
{

        fmtinstall('A', Aconv);
        fmtinstall('P', Pconv);
        fmtinstall('S', Sconv);
        fmtinstall('N', Nconv);
        fmtinstall('B', Bconv);
        fmtinstall('D', Dconv);
        fmtinstall('R', Rconv);
}

...

int
Pconv(Fmt *fp)
{
        char str[STRINGSZ], sc[20];
        Prog *p;
        int a, s;

        p = va_arg(fp->args, Prog*);
        a = p->as;
        s = p->scond;
        strcpy(sc, extra[s & C_SCOND]);
        if(s & C_SBIT)
                strcat(sc, ".S");
        if(s & C_PBIT)
                strcat(sc, ".P");
        if(s & C_WBIT)
                strcat(sc, ".W");
        if(s & C_UBIT)          /* ambiguous with FBIT */
                strcat(sc, ".U");
        if(a == AMOVM) {
                if(p->from.type == D_CONST)
                        sprint(str, "   %A%s    %R,%D", a, sc, &p->from, &p->to);
                else
                if(p->to.type == D_CONST)
                        sprint(str, "   %A%s    %D,%R", a, sc, &p->from, &p->to);
                else
                        sprint(str, "   %A%s    %D,%D", a, sc, &p->from, &p->to);
```

[44] http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/printk-formats.txt
[45] Universally unique identifier
[46] Globally Unique Identifier
[47] http://gmplib.org/manual/Formatted-Output-Strings.html

```
        } else
        if(a == ADATA)
                sprint(str, "   %A      %D/%d,%D", a, &p->from, p->reg, &p->to);
        else
        if(p->as == ATEXT)
                sprint(str, "   %A      %D,%d,%D", a, &p->from, p->reg, &p->to);
        else
        if(p->reg == NREG)
                sprint(str, "   %A%s    %D,%D", a, sc, &p->from, &p->to);
        else
        if(p->from.type != D_FREG)
                sprint(str, "   %A%s    %D,R%d,%D", a, sc, &p->from, p->reg, &p->to);
        else
                sprint(str, "   %A%s    %D,F%d,%D", a, sc, &p->from, p->reg, &p->to);
        return fmtstrcpy(fp, str);
}
```

(http://plan9.bell-labs.com/sources/plan9/sys/src/cmd/5c/list.c)

The Pconv() will be called if %P modifier in the format string will be met. Then it copies the string created using fmtstrcpy(). By the way, that function also uses other defined modifiers like %A, %D, etc.

The glibc has non-standard extension [48], allowing to define our own modifiers, but it is *deprecated*.

Let's try to define our own modifiers for Mac-address outputting and also for byte outputting in a binary form:

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

static int printf_arginfo_M(const struct printf_info *info, size_t n, int *argtypes)
{
        if (n > 0)
                argtypes[0] = PA_POINTER;

        return 1;
}

static int printf_output_M(FILE *stream, const struct printf_info *info, const void *const *args)
{
        const unsigned char *mac;
        int len;

        mac = *(unsigned char **)(args[0]);

        len = fprintf(stream, "%02x:%02x:%02x:%02x:%02x:%02x",
                        mac[0], mac[1], mac[2], mac[3], mac[4], mac[5]);

        return len;
}

static int printf_arginfo_B(const struct printf_info *info, size_t n, int *argtypes)
{
        if (n > 0)
                argtypes[0] = PA_POINTER;

        return 1;
}

static int printf_output_B(FILE *stream, const struct printf_info *info, const void *const *args)
{
        uint8_t val = *(int*)(args[0]);
```

[48] http://www.gnu.org/software/libc/manual/html_node/Customizing-Printf.html

```
        for (int i=7; i>=0; i--)
                fprintf(stream, "%d", (val>>i)&1);

        return 8;
}

int main()
{
        uint8_t mac[6] = { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55 };

        register_printf_function ('M', printf_output_M, printf_arginfo_M);
        register_printf_function ('B', printf_output_B, printf_arginfo_B);

        printf("%M\n", mac);
        printf("%B\n", 0xab);

        return 0;
};
```

[49]

This compiled with warnings:

```
1.c: In function 'main':
1.c:48:2: warning: 'register_printf_function' is deprecated (declared at /usr/include/printf.h
    :106) [-Wdeprecated-declarations]
1.c:49:2: warning: 'register_printf_function' is deprecated (declared at /usr/include/printf.h
    :106) [-Wdeprecated-declarations]
1.c:51:2: warning: unknown conversion type character 'M' in format [-Wformat]
1.c:52:2: warning: unknown conversion type character 'B' in format [-Wformat]
```

GCC is able to track accordance between modifiers in the printf-string and arguments in printf(), however, unfamiliar to it's modifiers which are present here, so it warns us about them.

Nevertheless, our program works:

```
$ ./a.out
00:11:22:33:44:55
10101011
```

2.5.6 atexit()

With the help of `atexit()` it is possible to add a function automatically called before each exit from your program. By the way, C++ programs use `atexit()` for adding global objects destructors.

Let's try to see:

```
#include <string>

std::string s="test";

int main()
{
};
```

In the assembly listing we will find constructor of the global object:

Listing 2.4: MSVC 2010

```
??__Es@@YAXXZ PROC                                    ; 'dynamic initializer for 's'', COMDAT
; Line 3
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG22192
```

[49] The base of example was taken from: http://codingrelic.geekhold.com/2008/12/printf-acular.html

```
        mov     ecx, OFFSET ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A
    ; s
        call    ??0?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QAE@PBD@Z ; std::
    basic_string<char,std::char_traits<char>,std::allocator<char> >::basic_string<char,std::
    char_traits<char>,std::allocator<char> >
        push    OFFSET ??__Fs@@YAXXZ                    ; 'dynamic atexit destructor for 's''
        call    _atexit
        add     esp, 4
        pop     ebp
        ret     0
??__Es@@YAXXZ ENDP                                      ; 'dynamic initializer for 's''

??__Fs@@YAXXZ PROC                                      ; 'dynamic atexit destructor for 's'',
    COMDAT
        push    ebp
        mov     ebp, esp
        mov     ecx, OFFSET ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A
    ; s
        call    ??1?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QAE@XZ ; std::
    basic_string<char,std::char_traits<char>,std::allocator<char> >::~basic_string<char,std::
    char_traits<char>,std::allocator<char> >
        pop     ebp
        ret     0
??__Fs@@YAXXZ ENDP                                      ; 'dynamic atexit destructor for 's''
```

Constructor, while constructing, also registers the object destructor in the `atexit()`.

2.5.7 bsearch(), lfind()

Handy functions for searching for something somewhere.

The difference between them is that lfind() just searches for data, buf bsearch() requires sorted data array, but may search faster [50] [51].

For example, search for a string in an array of pointers to a strings:

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

static int my_stricmp (const char *p1, const char **p2)
{
        //printf ("p1=%s p2=%s\n", p1, *p2); // debug
        return stricmp (p1, *p2);
};

int find_string_in_array_of_strings(const char *s, const char **array, size_t array_size)
{
        void *found=lfind (s, array, &array_size, sizeof(char*), my_stricmp);
        if (found)
                return (char**)found-array;
        else
                return -1; // string not found
};

int main()
{
        const char* strings[]={"string1", "hello", "world", "another string"};
        size_t strings_t=sizeof(strings)/sizeof(char*);

        printf ("%d\n", find_string_in_array_of_strings("world", strings, strings_t));
```

[50] by bisection method, etc

[51] the difference also is that bsearch() is present in [6], but lfind() is not, it is present only in POSIX and MSVC, but these functions are very straightforward to implement them on your own

```
        printf ("%d\n", find_string_in_array_of_strings("world2", strings, strings_t));
};
```

We need our own stricmp() because lfind() will pass a pointer to the string we looking for, but also to the place where pointer to the string is stored, not the string itself. If it would be an array of fixed-size strings, then usual stricmp() could be used here instead.

By the way, likewise comparison function is used for qsort().

lfind() returns a pointer to a place in an array where the `my_stricmp()` function was triggered returning 0. Then we compute a difference between the address of that place and the address of the beginning of an array. Considering pointer arithmetics(1.3.7), we get a number of elements between these addresses.

By implementing comparison function, we can search a string in any array. Example from OpenWatcom:

Listing 2.5: \bld\pbide\defgen\scan.c

```
int MyComp( const void *p1, const void *p2 ) {

    KeyWord     *ptr;

    ptr = (KeyWord *)p2;
    return( strcmp( p1, ptr->str ) );
}

static int CheckReservedWords( char *tokbuf ) {
    KeyWord     *match;

    match = bsearch( tokbuf, ReservedWords,
                     sizeof( ReservedWords ) / sizeof( KeyWord ),
                     sizeof( KeyWord ), MyComp );
    if( match == NULL ) {
        return( T_NAME );
    } else {
        return( match->tok );
    }
}
```

Here we have an array of structures sorted by the first field *ReservedWords*, like:

Listing 2.6: \bld\pbide\defgen\scan.c

```
typedef struct {
    char        *str;
    int         tok;
} KeyWord;

static KeyWord  ReservedWords[] = {
    "__cdecl",          T_CDECL,
    "__export",         T_EXPORT,
    "__far",            T_FAR,
    "__fortran",        T_FORTRAN,
    "__huge",           T_HUGE,
....
};
```

bsearch() searching for the string comparing it with the string in the first field of structure. bsearch() can be used here because the array is already sorted. Otherwise, lfind() should be used. Probably, an unsorted array can be sorted by qsort() before bsearch() usage, if you like the idea.

Likewise, it is possible to search anything anywhere. The example from BIND 9.9.1 [52]:

Listing 2.7: backtrace.c

```
static int
symtbl_compare(const void *addr, const void *entryarg) {
```

[52] https://www.isc.org/downloads/bind/

```
        const isc_backtrace_symmap_t *entry = entryarg;
        const isc_backtrace_symmap_t *end =
                &isc__backtrace_symtable[isc__backtrace_nsymbols - 1];

        if (isc__backtrace_nsymbols == 1 || entry == end) {
                if (addr >= entry->addr) {
                        /*
                         * If addr is equal to or larger than that of the last
                         * entry of the table, we cannot be sure if this is
                         * within a valid range so we consider it valid.
                         */
                        return (0);
                }
                return (-1);
        }

        /* entry + 1 is a valid entry from now on. */
        if (addr < entry->addr)
                return (-1);
        else if (addr >= (entry + 1)->addr)
                return (1);
        return (0);
}

...

        /*
         * Search the table for the entry that meets:
         * entry.addr <= addr < next_entry.addr.
         */
        found = bsearch(addr, isc__backtrace_symtable, isc__backtrace_nsymbols,
                        sizeof(isc__backtrace_symtable[0]), symtbl_compare);
        if (found == NULL)
                result = ISC_R_NOTFOUND;
```

Thus, it is possible to get rid of for() loop for enumerating elements each time, etc.

2.5.8 setjmp(), longjmp()

In some sense, it is C exceptions implementation.

`jmp_buf` is just a structure containing registers set, but most important are the address of current instruction and the stack pointer.

All the `setjmp()` does is just store register values into the structure. And all the `longjmp()` does is to restore the values.

It is often used for returning from deep places to outside, usually, in case of errors. Just like exceptions in C++.

For example, in the Oracle RDBMS, when some error occured, user sees a code and error message, the `longjmp()` is triggered somewhere from the deep in reality. For the same reason it is used in Bash.

It is even a more flexible mechanism than exceptions in other PL — it is not an issue to set return points in arbitrary points of a program and return to them when one needs to.

We can even imagine the `longjmp()` as a super-mega-goto which bypasses blocks, functions and restoring stack state.

However, it should be mentioned, all the `longjmp()` does is to restore the CPU registers. Allocated memory will remain so, no destructors as in C++ will be called. There are no such thing as RAII. On the other hand, since the part of the stack is just annuled, the memory allocated with the help of `alloca()`(2.1.1) will also be annuled.

2.5.9 stdarg.h

There are functions intended for variable arguments handling. At least the functions of the printf() and scanf() family are these.

Caveat #1

The variable of the type va_list may be used only once, if one need more, it should be copied:

```
va_list v1, v2;
va_start(v1, fmt);
va_copy(v2, v1);
// use v1
// use v2
va_end(v2);
va_end(v1);
```

2.5.10 srand() and rand()

PRNG[53] from the standard library has a very poor quality, besides, it is able to generate numbers only in 0..32767 limits. Avoid it.

2.6 C99 C standard

Full standard text: [6].

This standard is supported in GCC, Clang, Intel C++, but not in MSVC, and it is not known when it will be supported.
In order to turn it on in GCC, this compiler key should be added *-std=c99*.

[53] Pseudorandom number generator

Chapter 3

C++

3.1 Name mangling

In C an underscore symbol is to be prepended before each function name, so `function` may in fact have the following name in the object file: `_function`.

The C++ has operator overloading, so, several functions may share one name but different types. On the other hand, OS loader and linker are not aware of C++ and works with plain function names (or symbols). As a consequence, there is a need to encode function name, argument types, return value type, and probably a class name into one line.

For example, that is how the *box* class constructor is defined as:

```
box::box(int color, int width, int height, int depth)
```

... In the MSVC, conventions will have the following name: `??0box@@QAE@HHHH@Z` — for example, four consequtive *H* characters stand for the four consequtive argument of *int* type.

This is what is called *name mangling*.

That is why header files may contain `extern "C"`:

```
#ifdef  __cplusplus
extern "C" {
#endif

   void foo(int a, int b);

#ifdef  __cplusplus
}
#endif
```

This means that `foo()` is written in C, compiled as C function and will have the following name in the object files: `_foo`.

If you are to include that header in the C++ project, the compiler will treat its internal name as `_foo`. Without this directive, the compiler will look for the function named `?foo@@YAXHH@Z`.

Therefore, this directive is needed for linking C libraries to C++-projects.

And *ifdef* makes this directive visible only in C++.

More about *name mangling*: [19, 1.17].

3.2 C++ declarations

3.2.1 C++11: auto

When using STL, sometimes it is very boring to declare the type of iterator like:

```
for (std::list<int>::iterator it=list.begin(); it!=list.end(); it++)
```

The *it* can be deduced from `list.begin()`, so that is why starting from the C++11 standard, *auto* can be used instead:

```
for (auto it=list.begin(); it!=list.end(); it++)
```

3.3 C++ language elements

3.3.1 references

It is the same thing as pointers (1.3.7), but "safe", because it is harder to make a mistake while working with them [8, 8.3.2].

For example, a reference should be always be pointing to an object of corresponding type and can't be NULL [2, 8.6]. Even more, references cannot be changed, it's not possible to point to another object (reseat) [2, 8.5].

In [19, 1.7.1] it was demonstrated that on x86-level of code it is the same thing.

Just like pointers, *references* can be returned by functions, e.g.:

```
#include <iostream>

int& use_count()
{
        static int uc=1000; // starting value
        return uc;
};

void main()
{
        std::cout << ++use_count() << std::endl;
        std::cout << ++use_count() << std::endl;
        std::cout << ++use_count() << std::endl;
        std::cout << ++use_count() << std::endl;
};
```

This will print predictable:

```
1001
1002
1003
1004
```

If to see what was generated in assembly language, it can be seen that `use_count()` just returns a pointer to `uc`, and in the `main()` an increment of a value on this pointer is occuring.

3.4 Input/output

It is often necessary to output whole data structures into *ostream* while it is not handy to do output by each field. Sometimes it can be solved by adding the `ToString()` method to a class. Other solution is to make a free function for outputing like:

```
ostream& operator<< (ostream &out, const Object &in)
{
    out << "Object. size=" << in.size << " value=" << in.value << " ";

    return out;
};
```

Now it is possible to send objects right into *ostream*:

```
Object o1, o2;
...
cout << "o1=" << o1 << " o2=" << o2 << endl;
```

For the function's ability to access any class fields, it can be marked as *friend*. However, there is a point of view about not making them as *friend* for encapsulation reasons [12, Item 23 Prefer non-member non-friend functions to member functions].

3.5 Templates

Templates are usually necessary in order to make a class universal for several data types. For example, `std::string` is in fact `std::basic_string<char>`,
and `std::wstring` is `std::basic_string<wchar_t>`.

It is often done for data types like *float/double/complex* and even *int*. Some mathematical algorithm can be defined only once, but be compiled in several versions for all these data types.

Thus it is possible to define algorithms only once, but they will work for several data types.

Simplest examples are the *max, min, swap* functions working for any type, variables of which can be compared and assigned. Then you may want to write your own BigInt implementation, and if there is a comparison operator (`operator<`) is present, then written earlier *max/min* will work for the new class as well.

That is why lists and other containers in the STL are exactly templates: it can be said, they "embedds" the possibility of be united into list or collection to your class.

3.6 Standard Template Library

`std::string`, `std::list`, `std::map` and `std::set` internals are described in [19].

3.7 Criticism

- http://yosefk.com/c++fqa/
- Linus Torvalds: http://harmful.cat-v.org/software/c++/linus; http://yarchive.net/comp/linux/c++.html

Chapter 4

Other

What is stored in the binary (.o, .obj, .exe, .dll, .so) files?

Usually it is only data (global variables) and function bodies (including class methods).

There are no type information (classes, structures, typedefs(1.2.1)) there. This may helps in understanding how it works internally.

See also about name mangling: (3).

That is one of the serious decompilation problems — type information absence.

Read more about how everything is compiled into machine code: [19].

4.1 Error codes returning

The simplest way to indicate to caller about success is to return boolean value, *false* — in case of error, and *true* in case of success. A lot of such functions are present in the Windows API. And if one need to return more information, the error code may be left in TIB[1], from where it is possible to get it using GetLastError(). Or, in the UNIX-enviroments, to leave the error code in the global variable *errno*.

4.1.1 Negative error codes

Another interesting approach to pass more information in returning value. For example, in the manuals of the IBM DB2 9.1, we may spot this:

> Regardless of whether the application program provides an SQLCA or a stand-alone variable, SQLCODE is set by DB2 after each SQL statement is executed. DB2 conforms to the ISO/ANSI SQL standard as follows:
>
> If SQLCODE = 0, execution was successful.
> If SQLCODE > 0, execution was successful with a warning.
> If SQLCODE < 0, execution was not successful.
> SQLCODE = 100, "no data" was found. For example, a FETCH statement returned no data because the cursor was positioned after the last row of the result table.

[2] [3]

In the Linux kernel source code we may find this [4]:

Listing 4.1: arch/arm/kernel/crash_dump.c

```
/**
 * copy_oldmem_page() - copy one page from old kernel memory
 * @pfn: page frame number to be copied
 * @buf: buffer where the copied page is placed
 * @csize: number of bytes to copy
 * @offset: offset in bytes into the page
```

[1] Thread Information Block
[2] SQL codes
[3] SQL error codes
[4] http://lxr.free-electrons.com/source/arch/powerpc/kernel/crash_dump.c

```
 * @userbuf: if set, @buf is int he user address space
 *
 * This function copies one page from old kernel memory into buffer pointed by
 * @buf. If @buf is in userspace, set @userbuf to %1. Returns number of bytes
 * copied or negative error in case of failure.
 */
ssize_t copy_oldmem_page(unsigned long pfn, char *buf,
                         size_t csize, unsigned long offset,
                         int userbuf)
{
        void *vaddr;

        if (!csize)
                return 0;

        vaddr = ioremap(pfn << PAGE_SHIFT, PAGE_SIZE);
        if (!vaddr)
                return -ENOMEM;

        if (userbuf) {
                if (copy_to_user(buf, vaddr + offset, csize)) {
                        iounmap(vaddr);
                        return -EFAULT;
                }
        } else {
                memcpy(buf, vaddr + offset, csize);
        }

        iounmap(vaddr);
        return csize;
}
```

Take a look — a function many return both number of bytes and the error code. The ssize_t type is “signed” size_t, i.e., able to store negative values. ENOMEM and EFAULT are standard error codes from the errno.h.

4.2 Global variables

OOP[5] hype and other such things tells us that global variables is a bad thing, nevertheless, sometimes it is worth to use it (keeping in mind thread-awareness), e.g. for returning large amount of information from functions.

Thus, several C standard library functions are returned error code via global variable *errno*, which is not a global anymore in our time, but is stored in the TLS.

In Windows API the error code can be determined by calling the GetLastError(), which just takes a value from the TIB.

In the OpenWatcom compiler everything is stored in the global variables, so the very main function looks like:

Listing 4.2: bld\cg\c\generate.c

```
extern  void    Generate( bool routine_done )
/*****************************************/
/* The big one - here's where most of code generation happens.
 * Follow this routine to see the transformation of code unfold.
 */
{
    if( BGInInline() ) return;
    HaveLiveInfo = FALSE;
    HaveDominatorInfo = FALSE;
    #if ( _TARGET & ( _TARG_370 | _TARG_RISC ) ) == 0
        /* if we want to go fast, generate statement at a time */
        if( _IsModel( NO_OPTIMIZATION ) ) {
            if( !BlockByBlock ) {
                InitStackMap();
```

[5] Object-Oriented Programming

```
                BlockByBlock = TRUE;
            }
            LNBlip( SrcLine );
            FlushBlocks( FALSE );
            FreeExtraSyms( LastTemp );
            if( _MemLow ) {
                BlowAwayFreeLists();
            }
            return;
        }
    #endif

    /* if we couldn't get the whole procedure in memory, generate part of it */
    if( BlockByBlock ) {
        if( _MemLow || routine_done ) {
            GenPartialRoutine( routine_done );
        } else {
            BlkTooBig();
        }
        return;
    }

    /* if we're low on memory, go into BlockByBlock mode */
    if( _MemLow ) {
        InitStackMap();
        GenPartialRoutine( routine_done );
        BlowAwayFreeLists();
        return;
    }

    /* check to see that no basic block gets too unwieldy */
    if( routine_done == FALSE ) {
        BlkTooBig();
        return;
    }

    /* The routine is all in memory. Optimize and generate it */
    FixReturns();
    FixEdges();
    Renumber();
    BlockTrim();
    FindReferences();
    TailRecursion();
    NullConflicts( USE_IN_ANOTHER_BLOCK );
    InsDead();
    FixMemRefs();
    FindReferences();
    PreOptimize();
    PropNullInfo();
    MemtoBaseTemp();
    if( _MemCritical ) {
        Panic( FALSE );
        return;
    }
    MakeConflicts();
    if( _IsModel( LOOP_OPTIMIZATION ) ) {
        SplitVars();
    }
    AddCacheRegs();
    MakeLiveInfo();
    HaveLiveInfo = TRUE;
```

```
    AxeDeadCode();
    /* AxeDeadCode() may have emptied some blocks. Run BlockTrim() to get rid
     * of useless conditionals, then redo conflicts etc. if any blocks died.
     */
    if( BlockTrim() ) {
        FreeConflicts();
        NullConflicts( EMPTY );
        FindReferences();
        MakeConflicts();
        MakeLiveInfo();
    }
    FixIndex();
    FixSegments();
    FPRegAlloc();
    if( RegAlloc( FALSE ) == FALSE ) {
        Panic( TRUE );
        HaveLiveInfo = FALSE;
        return;
    }
    FPParms();
    FixMemBases();
    PostOptimize();
    InitStackMap();
    AssignTemps();
    FiniStackMap();
    FreeConflicts();
    SortBlocks();
    if( CalcDominatorInfo() ) {
        HaveDominatorInfo = TRUE;
    }
    GenProlog();
    UnFixEdges();
    OptSegs();
    GenObject();
    if( ( CurrProc->prolog_state & GENERATED_EPILOG ) == 0 ) {
        GenEpilog();
    }
    FreeProc();
    HaveLiveInfo = FALSE;
#if _TARGET & _TARG_INTEL
    if( _IsModel( NEW_P5_PROFILING ) ) {
        FlushQueue();
    }
#else
    FlushQueue();
#endif
}
```

4.3 Bit fields

A very popular thing in C, and also in programming generally.

For defining boolean values "true" or "false", it is possible to pass 1 or 0 in byte or 32-bit register, or in *int* type, but it is not very compact. Much more handy is to pass such values in specific bits.

For example, the standard C function findfirst() returns a structure about file it found where file attributes are encoded as:

```
#define _A_NORMAL 0x00
#define _A_RDONLY 0x01
#define _A_HIDDEN 0x02
#define _A_SYSTEM 0x04
#define _A_SUBDIR 0x10
```

```
#define _A_ARCH 0x20
```

Of course, it would not be very compact to pass each attribute by a *bool* variable.

Contrariwise, it is possible to use bit fields for passing flags into the function. For example, CreateFile() [6] from Windows API.

For flags specifying, in order not to make a typo and mess, they can be defined as:

```
#define FLAG1 (1<<0)
#define FLAG2 (1<<1)
#define FLAG3 (1<<2)
#define FLAG4 (1<<3)
#define FLAG5 (1<<4)
```

(The compiler will optimize it anyway).

These handy macros may be also used for specific bit/flag checking/setting/resetting:

```
#define IS_SET(flag, bit)       (((flag) & (bit)) ? true : false)
#define SET_BIT(var, bit)       ((var) |= (bit))
#define REMOVE_BIT(var, bit)    ((var) &= ~(bit))
```

On the other hand, one need to keep in mind that operation of isolation of each bit in the value of type *int* is usually costly for the CPU then *bool* type processing in 32-bit register . So if the speed is more crucial for you then memory footprint, you may try to use *bool*.

4.4 Interesting open-source projects worth for learning

4.4.1 C

- Go Compiler http://golang.org/doc/install/source
- Git https://github.com/git/git

4.4.2 C++

- LLVM http://llvm.org/releases/download.html
- Google Chrome http://www.chromium.org/developers/how-tos/get-the-code

[6] http://msdn.microsoft.com/en-us/library/windows/desktop/aa363858(v=vs.85).aspx

Chapter 5

GNU tools

5.1 gcov

(*Coverage*) Allows to show code lines executed and count:

Let's try simple example:

```
#include <stdio.h>

void f1(int in)
{
        int array[100][200];

        if (in>100)
        {
                printf ("Error!\n");
                return;
        };

        for (int i=0; i<100; i++)
                for (int j=0; j<200; j++)
                {
                        array[i][j]=in*i+j;
                };
};

int main()
{
        f1(12);
};
```

Compile it as (`-g` means adding debug information to the resulting executable file, `-O0` — absence of code optimization [1], the rest — gcov parameters):

```
gcc -std=c99 -g -O0 -fprofile-arcs -ftest-coverage gcov_test.c -o gcov_test
```

GCC inserts a statistics collection functions into the code. Of course, it slows down execution time. After the program execution, these files are generated: gmon.out, gcov_test.gcda, gcov_test.gcno.

Let's run gcov:

```
gcov gcov_test
```

We will got the text file gcov_test.c.gcov:

Listing 5.1: gcov_test.c.gcov

```
        -:    0:Source:gcov_test.c
        -:    0:Graph:gcov_test.gcno
        -:    0:Data:gcov_test.gcda
```

[1] It is very important because generated CPU instructions should be grouped and match C/C++ code lines. Optimization may distort this relation and gcov (and also gdb) will not be able to show source code lines correctly.

```
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:
        1:    3:void f1(int in)
        -:    4:{
        -:    5:        int array[100][200];
        -:    6:
        1:    7:        if (in>100)
        -:    8:        {
    #####:    9:                printf ("Error!\n");
        1:   10:                return;
        -:   11:        };
        -:   12:
      101:   13:        for (int i=0; i<100; i++)
    20100:   14:                for (int j=0; j<200; j++)
        -:   15:                {
    20000:   16:                        array[i][j]=in*i+j;
        -:   17:                };
        -:   18:};
        -:   19:
        1:   20:int main()
        -:   21:{
        1:   22:        f1(12);
        -:   23:};
        -:   24:
```

Strings marked as ##### was not executed. This can be particularly useful for tests writing.

Chapter 6

Testing

Testing is crucial. Simplest possible test is a program calling your functions and comparing their results with correct ones:

```
void should_be_true(bool a)
{
        if (a==false)
                die ("one of tests failed\n");
};
int main()
{
        should_be_true(f1(...)==correct_value1);
        should_be_true(f2(...)==correct_value2);
        should_be_true(f3(...)==correct_value3);
};
```

Tests should be work automatically (without human intervention) and be running as frequently as possible, ideally after each code change.

For tests writing, gcov (5) or any other coverage tool is very useful. Good test should test correctness of all functions, but also of all function parts.

Other testing articles and advices: [13].

Afterword

6.1 Questions?

Do not hesitate to mail any questions to the author: <dennis@yurichev.com>

Please, also do not hesitate to send me any corrections (including grammar ones (you see how horrible my English is?)), etc.

Acronyms used

STL Standard Template Library . . . 37

TLS Thread Local Storage . . . 22

TIB Thread Information Block . . . 51

RAII Resource Acquisition Is Initialization . . . 25

OS Operating System . . . i

PL Programming Language . . . 11

OOP Object-Oriented Programming . . . 52

PRNG Pseudorandom number generator . . . 47

MSVC Microsoft Visual C++ . . . 3

GCC GNU Compiler Collection . . . 3

POSIX Portable Operating System Interface . . . 37

CPU Central Processor Unit . . . 3

IDE Integrated development environment . . . 5

RISC Reduced instruction set computing . . . 3

IOCCC The International Obfuscated C Code Contest . . . 12

GUID Globally Unique Identifier . . . 41

UUID Universally unique identifier . . . 41

CRT C runtime library . . . 22

CSV Comma-separated values . . . 34

CPU Central processing unit . . . 3

Bibliography

[1] blexim. Basic integer overflows. Phrack, 2002. Also available as http://yurichev.com/mirrors/phrack/p60-0x0a.txt.

[2] Marshall Cline. C++ faq. Also available as http://www.parashift.com/c++-faq-lite/index.html.

[3] E. Dijkstra. Classics in software engineering. chapter Go to statement considered harmful, pages 27–33. Yourdon Press, Upper Saddle River, NJ, USA, 1979.

[4] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. Commun. ACM, 11(3):147–148, March 1968.

[5] Agner Fog. Optimizing software in C++. 2013. http://agner.org/optimize/optimizing_cpp.pdf.

[6] ISO. ISO/IEC 9899:TC3 (C C99 standard). 2007. Also available as http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf.

[7] ISO. ISO/IEC 9899:2011 (C C11 standard). 2011. Also available as http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1539.pdf.

[8] ISO. ISO/IEC 14882:2011 (C++ 11 standard). 2013. Also available as http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf.

[9] Donald E. Knuth. Structured programming with go to statements. ACM Comput. Surv., 6(4):261–301, December 1974. Also available as http://yurichev.com/mirrors/KnuthStructuredProgrammingGoTo.pdf.

[10] Donald E. Knuth. Computer literacy bookshops interview, 1993. Also available as http://yurichev.com/mirrors/C/knuth-interview1993.txt.

[11] John Lakos. Large-Scale C++ Software Design. 1996. http://www.amazon.com/Large-Scale-Software-Design-John-Lakos/dp/0201633620.

[12] Scott Meyers. Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition). 2005. http://www.amazon.com/Effective-Specific-Improve-Programs-Designs/dp/0321334876.

[13] Nicholas Nethercote. Good coding practices. Also available as http://njn.valgrind.org/good-code.html.

[14] Dennis Ritchie. about short-circuit operators. http://c-faq.com/misc/xor.dmr.html, 1995. [Online; accessed 2013].

[15] Dennis M. Ritchie. The evolution of the unix time-sharing system. 1979.

[16] Linus Torvalds. Linux kernel coding style. Also available as https://www.kernel.org/doc/Documentation/CodingStyle.

[17] Linus Torvalds. about typedef (fa.linux.kernel). http://yurichev.com/mirrors/C/ltorvalds_typedefs.html, 2002. [Online; accessed 2013].

[18] Linus Torvalds. about alloca() (fa.linux.kernel). http://yurichev.com/mirrors/C/ltorvalds_alloca.html, 2003. [Online; accessed 2013].

[19] Dennis Yurichev. An Introduction To Reverse Engineering for Beginners. 2013. Also available as http://yurichev.com/writings/RE_for_beginners-en.pdf.

Glossary

Integral type The data type that can be converted to a number type, like: int, short, char. 4, 10, 20, 21

Iterator The pointer to the current list or any other collection element, used for an elements enumerating. 1, 7, 8, 13, 48

Free function Function which is not a method of any class. 49

BigInt This is how a libraries for multiply precision numbers crunching are usually called, e.g. `http://gmplib.org/`. 41, 50

glibc The Linux standard library. 18, 42

Index