Заметки о языке программирования Си/Си++

Денис Юричев
<dennis@yurichev.com>

Версия этого текста (17 марта 2017 г.).
Возможно, более новая версии текста, а так же англоязычная версия, также доступна по ссылке http://yurichev.com/C-book.html
Вы также можете подписаться на мой twitter для получения информации о новых версиях этого текста, и т.д.: @yurichev, либо подписаться на список рассылки.

Оглавление

[1] Операционная Система

Введение

Сейчас, в 2013-м году, если некто желает написать 1) как можно более быстро работающую программу; 2) либо как можно более компактную для встраиваемых систем либо маломощных микроконтроллеров, то выбор очень ограниченный: Си, Си++ либо ассемблер. И альтернативы этим старым но популярным языкам, в обозримом будущем, пока что не видно.

О "чистом Си" также не стоит забывать, огромное количество больших программ продолжаются писаться на нем, например ядро Linux, ядра линейки Windows NT, Oracle RDBMS, и т.д..

0.1 Целевая аудитория

Этот сборник заметок предназначен не для начинающих, но и не для экспертов, а скорее для тех, кто хочет освежить свои знания по Си/Си++.

0.2 Благодарности

Андрей "herm1t" Баранович, Слава "Avid" Казаков, Tuta Muniz, Shell Rocket, Daniel Craig.

Глава 1

Общее для Си и Си++

1.1 Заголовочные файлы

Заголовочные файлы это описания некоего интерфейса, своего рода документация. Очень удобно в многооконном редакторе работать со своим кодом в одном окне, а в другом держать заголовочные файлы для справки. Поэтому очень полезно делать их похожими на справочник.

1.2 Определения и объявления

Разница между *определениями (declaration)* и *объявлениями (definition)*:

- *Определение (declaration)* определяет имя/тип переменной либо имя и типы аргументов а также возвращаемого значения ф-ции или метода. Обычно, определения перечисляются в заголовочных .h или .hpp-файлах, чтобы при компиляции отдельного .c или .cpp-файла, компилятор имел информацию об именах и типах внешних (обычно глобальных) переменных и ф-ций/методов.

  Типы данных также *определяются*.

- *Объявление (definition)* объявляет значение (обычно глобальной) переменной либо тело ф-ции/метода. Обычно это происходит в одном из .c или .cpp-файлах.

1.2.1 Определения в Си/Си++

Определения локальных переменных

Раньше, в Си можно было определять переменные только в начале ф-ции. А в Си++ — где угодно.

К тому же, нельзя было определять счетчик или итератор в for() (а в Си++ также можно было):

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

Новый стандарт C99(2.5.10) позволяет делать это.

static Если глобальные переменные (или ф-ции) определяются как *static*, так их область видимости ограничивается данным файлом. Но локальные переменные внутри ф-ции также можно определять как *static*, тогда эта переменная будет не локальной, а глобальной, но её область видимости будет ограничена только этой ф-цией.

Например:

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

К примеру, это помогло бы для реализации `strtok()`, ведь этой ф-ции что-то нужно хранить у себя между вызовами.

forward declaration

Как известно, в заголовочных файлах (headers) обычно содержатся декларации ф-ций, то есть, имя ф-ции, аргументы и типы, тип возвращаемого значения, но нет тела ф-ций. Так делается для того, чтобы компилятор мог знать, с чем имеет дело, не углубляясь в тонкости реализации ф-ций.

То же самое можно делать и для типов. Для того чтобы не включать при помощи #include файл с описаниями какого либо класса в другой заголовочный файл, можно обойтись указанием, что он вообще существует.

Например, вы работаете с комплексными числами и у вас где-то есть такая структура:

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

И, например, она определена в файле my_complex.h.

Безусловно, вам нужно включить этот файл, если вы собиретесь работать с переменными типа *complex*, с отдельными полями структуры. Но если вы описываете свои ф-ции для работы с этой структурой в отдельном заголовочном файле, то включать там my_complex.h не обязательно, компилятору достаточно просто знать что *complex* это структура:

```
struct complex;

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

Это позволяет увеличить скорость компиляции, а также бороться с циркулярными зависимостями, когда в двух модулях используются типы и ф-ции друг друга.

Частые ошибки

Чтобы объявить два указателя на *char*, можно, по инерции, написать:

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

Это не верно, потому что компилятор распознает это описание так:

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

... и определяет указатель на *char* и просто *char*.

Правильно так:

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

const

Определять переменные, аргументы ф-ций и методы классов в Си++ как const очень полезно, потому что:

- Документация кода — сразу видно что это элемент только для чтения.
- Защита от ошибок: в случае с глобальной переменной-const, при попытке записать в нее, сработает защита и процесс упадет. А если пытаться модифицировать const-аргумент ф-ции, компилятор выдаст ошибку.
- Оптимизация: компилятор, зная что элемент всегда "только для чтения", может сделать работу с ним эффективнее.

Желательно все аргументы ф-ции, которые вы не собираетесь модифицировать, определять как const. Например, ф-ция `strcmp()` ничего не меняет во входных аргументах, так что их обычно оба определяют как const. А, например, strcat() ничего не меняет во втором аргументе, но меняет в первом, поэтому обычно она определяется с *const* во втором аргументе.

Си++ В Си++, желательно все методы класса, которые не изменяют ничего в объекте, также определять как *const*.

const-методы класса называют также *аксессорами* (accessors), а не-const-методы — *манипуляторами* (manipulators) [11].

Типы данных

long double *float* это 32-битное число в формате IEEE 754, *double* это 64-битное, но FPU-сопроцессор в x86 оперирует 80-битными числами. Для них имеется тип *long double*, он поддерживается в GCC[1] но не в MSVC[2].

int В *int* обычно столько же бит сколько и разрядность регистров общего назначения процессора. Однако, в x86-64, для лучшей обратной совместимости, ширина *int* так и осталась 32 бита.

short, long и long long По крайней мере в MSVC и GCC *short* 16-битный, *long* — 32-битный, а *long long* — 64-битный.

Во избежании путанницы, в *stdint.h* (по крайней мере в C99) имеются типы *int8_t*, *uint8_t*, *int16_t*, *uint16_t*, *int32_t*, *uint32_t*, *int64_t*, *uint64_t*.

bool bool есть в Си++, но также он есть и в Си, начиная с C99(2.5.10) (stdbool.h).
И в MSVC и в GCC, bool занимает 1 байт.
В Windows API принят тип BOOL, это синоним *int*.

Знаковые или беззнаковые? Знаковые типы (int, char) используются куда чаще беззнаковых (unsigned int, unsigned char) [3].

Но с точки зрения документации кода, если вы определяете переменную, которая никогда не будет хранить отрицательное значение, в т.ч., индексы массивов, наверное лучше применять беззнаковый тип. Например, в LLVM очень часто используется *unsigned* там где обычно можно было бы использовать *int*.

Если вы работаете с байтами, например, с байтами в памяти, то наверное лучше применять именно *unsigned char* .

К тому же, это может немного защититься от ошибок связанных с integer overflow [1].

В качестве очень простого примера:

```
#define MAX_BUFFER_SIZE 1024

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

Если *size* будет, например, -1, то malloc() вызовется с аргументом 0xffffffff (это 4294967295). Конечно, нужно было бы добавить вторую проверку: *if (size<0)*, но такая проверка выглядит здесь абсурдной.

Таким образом, здесь нужно было бы применить *unsigned*, либо даже тип *size_t*. *size_t* определяет тип, достаточно большой, способный хранить размер любого, сколько угодно большого блока памяти. На 32-битных архитектурах это обычно *unsigned int*, а на 64-битных это *unsigned int64*.

char или uint8_t вместо int? Может показаться что если какая-то переменная всегда будет в пределах $0..100$, то незачем выделять под нее 32-битный *int*, а можно обойтись типом *char* или *unsigned char*. К тому же, такая переменная будет занимать в памяти в 4 раза меньше.

Это не так. Из-за выравнивания по 4-байтной границе (а в 64-битных архитектурах — по 8-байтной), определяемые переменные типа *char*, занимают столько же места сколько и *int*.

Конечно, компилятор мог бы отводить под *char* только один байт, но тогда CPU[4] тратил бы больше времени на обращение к "невыровненным" по границе байтам.

Работа с отдельными байтами может быть "дороже" и медленнее чем работа с 32-битными или 64-битными значениями потому что регистры CPU обычно имеют ту же ширину что и разрядность процессора. И даже более того, RISC[5]-процессоры (например ARM) вообще могут быть неспособны работать с отдельными байтами внутренне, потому что имеют только 32-битные регистры.

[1] GNU Compiler Collection
[2] Microsoft Visual C++
[3] Стандартном не закреплен тип *char*, но в GCC и MSVC по умолчанию это именно знаковый тип. При желании в GCC можно изменить это задав ключ `-funsigned-char` а в MSVC ключ `/J`.
[4] Central processing unit
[5] Reduced instruction set computing

Таким образом, если вы раздумываете над типом для локальной переменной, то *int/unsigned int* может быть лучше.

С другой стороны, переменные каких типов лучше использовать в структурах? Это вопрос поиска баланса между скоростью и компактностью. С одной стороны, можно отвести *char* под небольшие переменные, под флаги, под enum, и т.д., но не следует забывать, что доступ к этим переменным будет чуть медленнее. С другой стороны, под все переменные можно отводить *int*, тогда работа со структурой будет быстрее, но она будет занимать в памяти больше места.

Например:

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

Если скомпилировать это с упаковкой полей по 1-байтной границе, то доступ к *some_flags* в памяти будет возможно даже быстрее чем доступ к *ptr*, потому что первое поле выровнено по 4-байтной границе, а второе нет.

А если компилировать это с упаковкой полей по умолчанию, то компилятор отведет под первое поле 4 байта и смещение у второго поля будет +4.

Резюмируя: если компактность и экономия памяти для вас важнее скорости, тогда нужно использовать *char*, *uint16_t*, и т.д..

x86-64 или AMD64 На новых 64-битных x86-процессорах, тип *int/unsigned int* оставили 32-битным, вероятно, в целях совместимости. Так что если вы хотите использовать 64-битные значения, можно использовать *uint64_t* или *int64_t*.

А указатели теперь, конечно, 64-битные.

typedef

`typedef` вводит *синоним* для типа данных. Часто это используется для структур, чтобы каждый раз не писать *struct* перед её именем, например:

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

Такого очень много в "заголовочных" файлах в Windows SDK (Windows API).

Тем не менее, *typedef* также можно использовать не только для структур, но и для обычных, интегральных типов например:

```
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);
```

Как видно, *typedef* здесь может помочь в документировании исходного кода, так он легче читается.

Например, тип *time_t* (время в формате UNIX time , то что возвращает стандартная функция localtime() , например), это на самом деле обычное 32-битное число, но этот тип определен в *time.h* обычно так:

```
typedef long __time32_t;
```

Здесь вполне можно было бы использовать директиву препроцессора `#define` (многие так и делают), но это хуже с точки зрения обработки ошибок во время компиляции.

Критика typedef Тем не менее, такой известный и опытный программист как Линус Торвальдс, против использования typedef: [17].

1.2.2 Описания (definitions)

Определение строк

Последовательности символов используемые в строках

\0	0x00	нулевой байт
\a	0x07	звонок [6]
\t	0x09	табуляция
\n	0x0A	line feed (LF)
\r	0x0D	carriage return (CR)

Разница между LF и CR в том, что в старых матричных принтерах, LF означал протягивание бумаги на одну строку вниз, а CR перевод каретки влево до края бумаги. Так что в принтер нужно было передавать оба символа.

Вывод CR без LF дает возможность перезаписывать текущую строку в консоли:

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

Это часто используется например в архиваторах для отображения текущего статуса и wget.

В Си и UNIX принят LF как символ новой строки.

В DOS и затем в Windows — CR+LF.

Строка определенная в нескольких строках Малоизвестная возможность Си, длинные строки можно объявлять так:

```
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]);
```

Это отдаленно напоминает “here document”[7] в UNIX-шеллах и Perl.

1.3 Элементы языка

1.3.1 Комментарии

Их иногда удобно вставлять прямо в вызов ф-ции, чтобы где-то на виду держать пометку, что означает некий аргумент:

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

Целый блок кода можно откомментировать при помощи #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);
```

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

[8] директива препроцессора

```
        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);
```

Это может быть удобнее чем традиционный способ потому что текстовый редактор или IDE[9] в этом случае не "сломает" отступы при выравнивании.

1.3.2 goto

Использование *goto*[10] считается плохим тоном и вредным вообще [4] [3], тем не менее, использование его в разумных дозах [9] может облегчить жизнь.

Частый пример, это выход из функции:

```
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;
};
```

Более сложный пример:

```
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;

        ...
```

[9] Integrated development environment
[10] statement

```
close_file_cleanup_and_exit:
        fclose(f);

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

Если в данных примерах отказаться от *goto*, то придется вызывать *free()* и *fclose()* перед каждым выходом из функции (*return*), что здорово замусорит весь код.

Использование *goto* в таких случаях одобряется, например, в [16].

1.3.3 for

В for(), как известно, три выражения: первое вычисляется перед началом всех итераций, второе вычисляется перед каждой итерацией, третье — после каждой итерации.

И конечно же, там можно указывать что-то отличное от обычного счетчика.

Засада #1

Если написать такое:

```
#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;
};
```

... то это наверное будет ошибкой: `strlen(s)` будет вызываться перед каждой итерацией — такой код генерирует MSVC 2010. Впрочем, GCC 4.8.1 вызывает `strlen(s)` только один раз, в начале цикла.

Запятая

Запятая [6, 6.5.17] — не самая понятная для всех штука в Си, однако, их очень удобно использовать в определениях в for().

Например, может пригодится использовать в цикле два счетчика или итератора одновременно. Пусть счетчик просто отсчитывает от 0, прибавляя 1 при каждой итерации, а итератор указывает на элемент в списке:

```
#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;
};
```

Это выдаст предсказуемый результат:

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

Но к сожалению, определять счетчики или итераторы вместе с разными типами в теле самого for() вот так нельзя:

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

Тем не менее, переменные одного типа определять можно:

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

continue

continue это безусловный переход на конец тела цикла.

Это может быть очень полезно, например, в подобном коде:

```
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();
                };

        };
};
```

... всё это можно легко заменить на более опрятное:

```
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

Вместо коротких *if* желательно использовать ?:, например:

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

...

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

Си++: определения переменных в if()

Это доступно как минимум в стандарте C++03:

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

Точно также их можно определять и в switch().

1.3.5 switch

Иногда можно устать писать одно и то же:

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

А вот это нестандартное расширение GCC [11] может немного всё упростить:

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

Так что если в планах имеется использовать только компилятор GCC, то можно делать так.

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

Объявление переменных внутри switch()

Этого делать нельзя, но зато можно открывать новый блок и объявлять их уже там (в Си++ или начиная с C99):

```
switch(...)
{
        case 0:
                {
                        int x=1,y=2;
                        fn1(x, y);
                };
                break;
        case 1:
        case 2:

        ...
};
```

1.3.6 sizeof

Обычно, sizeof() применяют к интегральным типам либо к структурам, тем не менее, его можно применять и к массивам, к примеру:

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

В противном случае, если указывать длину массива (1024) в двух местах (в определении buf и как второй аргумент snprintf()), то и изменять это значение придется каждый раз в обоих местах, а об этом легко забыть.

Если нужны wide-строки, то sizeof() можно применять к *wchar_t* (который, на самом деле, 16-битный тип данных *short*):

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

sizeof() возвращает длину в байтах, так что здесь он будет равен $1024 * 2$, т.е., 2048. Но мы можем разделить это значение на длину одного элемента массива (*wchar_t*) в байтах (2), чтобы получить количество элементов в массиве (1024).

sizeof() можно применять и к массивам структур:

```
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) — это размер всего массива структур в байтах. sizeof(struct phonebook_entry) — это размер одной структуры в байтах. Делением мы узнаем количество структур в массиве.

1.3.7 Указатели

Как однажды сказал Дональд Кнут в интервью [10], то как в Си устроены указатели, это является очень удачной инновацией в языках программирования по тем временам.

Итак, определимся с терминологией. Указатель это просто адрес какого-то элемента в памяти. Указатели настолько популярны, потому что в какую-то функцию намного проще передать просто адрес объекта в памяти, вместо того чтобы передавать весь объект — ведь это абсурдно.

К тому же, вызываемая функция, например, обрабатывающая ваш массив данных, просто изменит что-то в нем, вместо того чтобы возвращать новый, измененный массив данных, что тоже абсурдно.

Возьмем простой пример. Стандартная функция *strtok()* делит строку на подстроки, используя заданный символ как разделитель. К примеру, мы можем подать на вход строку `The quick brown fox jumps over the lazy dog` и задать пробел в качестве разделителя.

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

Мы в итоге получим на выходе:

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

Что тут в реальности происходит, это то что ф-ция *strtok()* просто находит в заданной строке следующий пробел (либо иной заданный разделитель), записывает туда 0 (что по соглашениям текстовых строк в Си является концом строки) и возвращает указатель на это место.

В качестве недостатка *strtok()* можно отметить, что эта ф-ция "портит" входную строку, записывая нули на месте разделителей.

Но вот что важно заметить: никакие строки или подстроки не копируются в памяти. Входная строка остается там же где и лежала.

В *strtok()* передается только указатель на нее, или, её адрес.

Эта ф-ция затем, после того как записывает 0, возвращает *адрес* каждого следующего "слова".

Адрес "слова" затем подается на вход в *printf()*, где происходит его вывод на консоль.

N.B. В исходнике присутствует и некорректное определение *str*.

Оно тем некорректное что в Си строка имеет тип *const char**, то есть, распологается в константном сегменте данных, защищенным от записи.

Если так сделать, то *strtok()* не сможет модифицировать входную строку записывая туда нули и процесс "упадет".

Так что, в нашем примере, строка выделяется как массив *char* а не массив *const char*.

Обобщая, скажем что работа со строками в Си происходит только лишь используя адреса этих строк.

К примеру, ф-ция сравнения строк *strcmp()* берет на вход два адреса двух строк и по одному символу сравнивает их. Было бы очень абсурдно копировать куда-то эти две строки лишний раз, чтобы *strcmp()* обработала их.

Трудность понимания указателей в Си связана с тем, что указатель это "часть" объекта. Указатель на строку, это не сама строка. Сама строка еще должна где-то в памяти хранится, под нее нужно перед этим выделять место, и т.д..

В ЯП[12] более высокого уровня, объект и указатель на него могут быть представлены как единое целое, что облегчает понимание.

Впрочем, это не значит что в этих ЯП строки и иные объекты неразумно копируются много раз при передаче в другие функции — там точно так же как и в Си используются указатели, но просто эта механика скрыта от программиста.

Передача значения в ф-цию также называется "call by value" или "pass by value" в то время как передача указателя на объект называется "call by reference" или "pass by reference".

Синтаксический сахар для array[index]

Ради упрощения, можно сказать что в Си нет массивов вообще, а есть только синтаксический сахар для выражений вроде *array[index]*.

К примеру, возможно вы видели такой трюк:

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

Это выдаст 'l'.

Это происходит, потому что любое выражение *a[i]*, на самом деле преобразовывается в **(a+i)* [6, 6.5.2/1]. *3["hello"]* преобразовывается в **(3+"hello")*, а *"hello"* это просто указатель на массив символов, типа *const char**.

3+"hello" это в итоге указатель на часть строки, то есть, *"lo"*. А **("lo")* это символ 'l'. Вот почему это работает.

Но так врядли стоит писать, если вы конечно не готовите программу на конкурс IOCCC[13][14]. Так что я привел этот пример, чтобы наглядно показать, что выражения вроде *a[i]* это синтаксический сахар.

При некотором упорстве, в Си вообще можно обойтись без индексации массивов, хотя выглядеть это будет не очень эстетично.

Кстати, так легко понять как работают отрицательные индексы массивов. *a[-3]* просто преобразуется в **(a-3)*, так адресуется элемент лежащий перед самим массивом. И хотя это вполне возможно, так можно делать только если вы точно знаете, что вы делаете.

Еще один трюк связанный с негативными индексами: например, когда вы привыкли адресовать массивы начиная не с 0, а с 1 (как в FORTRAN), тогда можно сделать такое:

```
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
};
```

Хотя снова нельзя с уверенностью сказать что использование таких трюков оправдано.

Так что в Си массив это, в каком-то смысле, это просто место в памяти под массив плюс указатель, указывающий на него.

Вот почему имя массива в Си можно считать за указатель:

Если вы объявите глобальную переменную *int a[10]*, то *(a)* будет иметь тип *int**. Позже, когда где-то в коде вы укажете *x=a[5]*, выражение будет преобразовано в *x=*(a+5)*. От начала массива (то есть, первого элемента массива), будет отсчитано 5 элементов, затем оттуда прочитается элемент для записи в *(x)*.

Арифметика указателей

Простой пример:

```
#include <stdio.h>

struct phonebook_entry
```

[12] Язык Программирования
[13] The International Obfuscated C Code Contest
[14] http://www.ioccc.org/

```
{
        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++)
                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);
};
```

Мы объявляем глобальный массив из структур. Если скомпилировать это в GCC с ключом *-S* либо в MSVC с ключом */Fa*, мы увидим в листинге на ассемблере то, как компилятор расположил эти строки.

Расположил он их как линейный массив указателей на строки, вот так:

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

Ф-ции *dump1()* и *dump2()* эквивалентны.

Но в первой счетчик *(i)* начинается с 0 и к нему прибавляется 1 на каждой итерации.

Во второй ф-ции итератор *(i)* указывает на начало массива и затем, к нему прибавляется длина структуры (а не 1 байт, как можно поначалу ошибочно подумать), таким образом, на каждой итерации, *(i)* указывает на следующий элемент массива.

Указатели на функции

Часто используются для callback-в.

Из-за того что можно напрямую задавать адрес функции, в embedded-программировании, так можно сделать переход по нужному адресу:

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

Впрочем, нужно помнить, что это не совсем аналог безусловного перехода, потому что в стеке сохраняется адрес возврата, может быть что-то еще.

1.3.8 Операторы

==

Очень неприятные ошибки возникают если в условии *if(a==3)* опечататься и написать *if(a=3)*. Ведь выражение *a=3* "возвращает" 3, а 3 это не 0, поэтому условие *if()* всегда будет срабатывать.

Раньше, для защиты от подобных ошибок, была мода писать наоборот: *if(3==a)*, таким образом, если опечататься, выйдет *if(3=a)*, компилятор тут же выдаст ошибку.

Тем не менее, в наше время, компиляторы обычно предупреждают если написать *if(a=3)*, так что, наверное, менять местами элементы выражения уже не обязательно.

Short-circuit evaluation и артефакт приоритетов операций

Разберем что такое *short-circuit* [15]evaluation.

Это когда в выражении *if(a && b && c)*, часть *(b)* будет вычисляться только если *(a)* — истинна, а *(c)* будет вычисляться только если *(a)* и *(b)* — оба истинны. И вычисляться они будут именно в таком порядке, как указано.

Иногда можно встретить подобное: *if (p!=NULL && p->field==123)* — и это совершенно правильно. Поле *field* в структуре, на которую указывает *(p)*, будет вычисляться только если указатель *(p)* не равен *NULL*.

То же касается и операции "или", если в выражении *if (a || b || c)* подвыражение *(a)* будет "истинно", остальные вычисляться не будут.

Это может быть удобно для вызова нескольких ф-ций: *if (get_flagsA() || get_flagsB() || get_flagsC())* — если первая или вторая ф-ция вернет *true*, остальные даже не будут вызываться.

Эта особенность есть не только в Си/Си++ [16].

Когда-то давно [14], в языках B и BCPL (предтечи Си) не было операторов && и ||, но чтобы реализовать в них *short-circuit evaluation*, приоритет операций & и | сделали больше, чем, например, у ^ или + [17].

Это позволяло писать что-то вроде *if (a==1 & b==c)* используя & вместо &&. Вот откуда взялся этот артефакт в приоритетах.

Так что, нередкая ошибка это забывать о высоком приоритете этих операций и писать, например, *if (a&1==0)*, в то время как это нужно брать в скобки: *if ((a&1)==0)*.

! и ~

~ (тильда) это побитовое инвертирование всех бит в значении.

Эта операция часто используется для инвертирования результатов действия ф-ций. Например, strcmp() в случае равенства строк возвращает 0. Поэтому можно писать:

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

... вместо `if (strcmp (...)==0)`.

Также, два подряд восклицательных знака применяется для трасформирования любого значения в тип bool по правилу: 0 — false (0); не ноль — true (1).

Например:

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

[15]дословный перевод на русский: "короткое замыкание"

[16]Здесь список ЯП где присутствует *short-circuit evaluation*https://en.wikipedia.org/wiki/Short-circuit_evaluation. Кстати, хотя это и не про Си, но все же интересно: в bash если писать *cmd1 && cmd2 && cmd3*, то каждая следующая команда будет исполняться только если предыдущая закончилась с успехом. Это также *short-circuit*.

[17]Приоритет операций в Си++: http://en.cppreference.com/w/cpp/language/operator_precedence

Или:

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

А также:

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

1.3.9 Массивы

В C99(2.5.10) можно передавать массив в аргументах ф-ции.

Собственно, массив байт можно было передавать и в более старых стандартах Си, кодируя байты в строке, включая ноль, примерно так (узнать, встречается ли байт (c) в массиве байт)(2.2.5):

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

Байты после ноля нормально кодируются.

Но в C99 теперь можно передавать массив значений других типов, например 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);
```

Поиск в массиве можно реализовать при помощи ф-ций bsearch() или lfind()(2.5.7), поиск и вставку при помощи lsearch() [18].

Инициализация

В GCC можно [19] инициализировать части массивов:

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

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

... но это нестандартное расширение.

1.3.10 struct

В C99(2.5.10) можно инициализировать отдельные поля структур. Пропущенные будут заполнены нулями. Такого очень много в ядре Linux.

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

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

И даже более того, можно создавать структуру прямо в аргументах ф-ции, например:

[18] работает также как и lfind(), но при отсутствии искомого элемента, добавляет его в массив

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

```
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 });
};
```

Точно также стурктуру можно и возвращать из ф-ции:

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

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

Помимо всего прочего, о структурах также много есть в разделе "Объектно-ориентированное программирование в Си"(2.3.3).

Расположение полей в структурах (cache locality)

В современных x86-микропроцессорах (как Intel, так и AMD) имеется кеш-память разных уровней. Самая быстрая кеш-память (L1), разделена на 64-байтные элементы (кеш-линии) и любое обращение к памяти заполняет сразу всю линию [5].

Можно сказать, что любое обращение к памяти (по выровненным адресам) подтягивает в кеш сразу 64 байта.

Поэтому, если некая структура данных имеет размер более 64-х байт, очень важно разделить её на две части: наиболее востребованные поля и менее востребованные. Самые востребованные поля желательно разместить в пределах первых 64-х байт.

Это же касается и классов в Си++.

1.3.11 union

`union` часто используется, когда в каком-то месте структуры нужно хранить разные типы на выбор. К примеру:

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

Такой `union` позволяет хранить одну из этих трех переменных на выбор. Занимать он будет места столько же, сколько максимальный элемент (double) — 8 байт.

`union` часто используют для обращения к какому-то типу данных как к другому.

Например, как известно, каждый XMM-регистр в SSE может представлять собой 16 байт, 8 16-битных слов, 4 32-битных слова, 2 64-битных слова, 4 float-значения и 2 double-значения. Так можно описать его:

```
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[...]
```

Это также очень удобно использовать вместе со структурой, где поля имеют битовую гранулярность. Как флаги x86-процессора:

```
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;
```

Можно таким образом загрузить флаги как 32-битное значение в поле *flags*, а затем из поля *(s)* обращаться к отдельным битам. Либо наоборот, модифицировать биты, затем прочитать поле *flags*. Такого очень много в исходниках ядра Linux.

Еще один пример использования *union* для определения порядка байтов (endianness):

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

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

[20]

tagged union

Это union плюс флаг (tag), определяющий тип union. К примеру, если нам нужна какая-то переменная, которая может быть как числом, так и числом с плавающей точкой, так и текстовой строкой (как переменные в динамически-типизированных ЯП [21]), то мы можем объявить такую структуру:

```
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;
} variable;
```

Суммарная длина такой структуры будет $8 + 4 = 12$ байт. В любом случае, это компактнее, чем выделять поля для переменной каждого возможного типа.

Начиная с C11 [7], (u) можно не указывать, это называется "анонимный 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;
```

... и обращаться к полям union просто как к variable.i, variable.d, и т.д..

Тэггированные указатели

Вернемся к примеру объявления переменной, которая может быть как числом, так и числом с плавающей запятой, так и текстовой строкой. Самый большой тип — double (8 байт), следовательно, имея много таких блоков в памяти рядом, указатель на каждый блок будет всегда выровнен по 8-байтной границе. И даже более того, в malloc() glibc всегда выделяет блоки по 8-байтной границе. Следовательно, указатель на подобный блок всегда будет иметь нули в трех младших битах. А раз так, то эти младшие биты можно использовать для чего-то. Одна из возможностей, это хранить там тип *union*. У нас всего три различных типа переменных, для хранения числа в диапазоне 0..2 нужно только 2 бита.

Это называется тэггированный указатель (tagged pointer). Так можно сэкономить на памяти и убрать из структуры *enum* указывающий на тип.

[20] пример взят отсюда: http://stackoverflow.com/questions/1001307/detecting-endianness-programmatically-in-a-c-program
[21] в Visual Basic это называется также "variant type"

Это очень популярно в LISP-интерпретаторах и компиляторах, потому что атомы в LISP-е это как раз вот такие структуры, описывающие каждую переменную, их может быть очень много, и использование младших бит указателя здорово экономит память.

Точно так же, любая другая информация может хранится в этих битах указателя.

В качестве негативной стороны, нужно всегда помнить о том что указатель теперь не совсем указатель, а содержит еще информацию. Отладчики не смогут работать с такими указателями корректно.

1.4 Препроцессор

Препроцессор обрабатывает директивы начинающиеся с # — #define, #include, #if, и т.д..

Листинг 1.1: "или"

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

1.5 Стандартные для компиляторов и ОС значения

- _DEBUG — отладочная сборка.
- NDEBUG — неотладочная (release) сборка.
- __linux__ — ОС Linux.
- _WIN32 — ОС Windows. Присутствует как и в x86-проектах, так и в x64. Отсутствует в Cygwin.
- _WIN64 — Присутствует в x64-проектах для ОС Windows.
- __cplusplus — присутствует в Си++ проектах.
- _MSC_VER — компилятор MSVC.
- __GNUC__ — компилятор GCC.
- __APPLE__ — компиляция под устройства Apple.
- __arm__ — компиляция под процессор ARM (GCC, Keil).
- __ppc__ — компиляция под PowerPC 32-bit.
- __ppc64__ — компиляция под PowerPC 64-bit.
- __LP64__ или _LP64 — GCC: компиляция под 64-битный режим [22].

Так можно писать разные участки кода для разных компиляторов и ОС.

Прочие макросы разных ОС: http://sourceforge.net/p/predef/wiki/OperatingSystems/

1.5.1 Еще стандартные макросы препроцессора

__FILE__, __LINE__, __FUNCTION__ — соответственно, имя текущего файла, номер текущей строки и имя текущей ф-ции.

Для того чтобы получить значения __FILE__ и __FUNCTION__ в UTF-16, можно воспользоваться следующим хаком:

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

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

[22] Дословно: long pointer, т.е., указатель требует 64 бита для хранения

1.5.2 "Пустой" макрос

Всем известны макросы не объявляющие никаких значений, например *_DEBUG*. Обычно, только проверяется наличие его или отсутствие. Вот еще пример полезного "пустого" макроса:

В заголовочных файлах Windows API мы можем увидеть такое:

```
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 и OPTIONAL — это "пустые" макросы объявленные так:

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

Для компилятора они никакой информации не несут, они предназначены только для документирования, показать, какие параметры ф-ций зачем нужны.

1.5.3 Частые ошибки

#1

К примеру, вы хотите создать макрос для возведения числа в квадрат:

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

Это ошибка, потому что выражение *square(a+b)* в итоге "развернется" в $a + b * a + b$, что, разумеется, совсем не то что хотелось. Поэтому в определнии макроса все переменные, и сам макрос, нужно "изолировать" скобками:

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

Пример из файла minmax.h из MinGW:

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

#2

Если вы где-то определяете какую-то константу:

```
#define N 1234
```

... затем где-то дальше переопределяете её снова, то компилятор промолчит, и это приведет к трудновыявляемой ошибке.

Поэтому константы желательнее определять как глобальные переменные с модификатором *const*.

1.6 Предупреждения компилятора

Стоит ли постоянно держать включенным ключ `-Wall` в GCC или `/Wall` в MSVC, то есть, чтобы выводить все возможные предупреждения (warnings)? Да, однозначно стоит, так можно заранее найти мелкие ошибки. Можно даже в GCC включить `-Werror` или `/WX` в MSVC — тогда все предупреждения будут трактоваться как ошибки.

1.6.1 Пример #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));
};
```

Автор "забыл" дописать *return* в ф-ции f1(). Тем не менее, GCC 4.8.1 компилирует этот пример молча.

Это связано с тем что в стандарте Си ([6, 6.9.1/12]) и в Си++ ([8, 6.6.3/2]) допустимо если ф-ция не возвращает значение.

При запуске мы увидим это:

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

Откуда взялось число 14? Это то что вернула ф-ция printf() вызванная из f1(). Возвращаемые результаты ф-ций интегральных типов остаются в регистре EAX/RAX. В ф-ции main() берется значение из регистра EAX/RAX и передается дальше во второй printf() [23].

Если компилировать с опцией `-Wall`, GCC скажет:

```
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]
 };
 ^
```

... хотя всё равно скомпилирует.

MSVC 2010 генерирует код, работающий точно также, хотя и выводит предупреждение:

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

Как видно, ошибка почти критическая, вызванная, можно сказать, опечаткой, но предупреждения компилятора либо не видно вовсе, либо можно и не заметить.

1.6.2 Пример #2

В Си-стандарте C99 появился тип *bool* , и согласно этому стандарту, он должен быть достаточным, чтобы хранить там по крайней мере один бит. В GCC bool это байт.

Если GCC не находит объявления некоей используемой ф-ции, он по умолчанию считает его возвращаемый тип за *int*, о чем предупреждает.

Далее, представим что имеем два файла:

[23] Больше о том, как возвращаются результаты ф-ций через регистры, можно почитать в [19]

Листинг 1.2: file1.c

```
bool f1()
{
        ...

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

Листинг 1.3: file2.c

```
...

if (f1())
        do_something();

...
```

При компиляции file2.c, у GCC нет информации о f1() и он считает что возвращается *int*. При компиляции file1.c GCC знает что нужно вернуть bool, но достаточно просто байт. Переменные интегральных типов возвращаются через регистр EAX или RAX x86-процессоров [24], так что GCC генерирует код, который выставляет только младший байт этого регистра (AL) в 1 или 0, а остальную часть регистра он может вообще не трогать, и там может оставаться случайный мусор от исполнения другого кода. Так что сгенерированный код в f1() может возвращать false записывая 0 в младший байт регистра EAX/RAX, тогда как в остальных битах будет мусор. С точки зрения file2.c, где принимается что f1() возвращает *int*, возвращаемое число может выглядеть так: 0x??????00, где *?* — случайные биты. Поэтому, даже если f1() возвращает false, условие if() может срабатывать почти всегда.

Автору этих строк удалось найти такую ошибку только заглянув в отладчик и ассемблерные листинги этих ф-ций, и пришлось потратить несколько часов.

Вариация этой ошибки:

Листинг 1.4: file1.c

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

Листинг 1.5: file2.c

```
...

uint64_t tmp=f1();

...
```

Если компилятор будет считать тип значения возвращаемого f1() за int, то 64-битное значение будет "обрезаться" до 32-битного (потому что int в 64-битной среде, вероятно для лучшей совместимости, оставили 32-битным).

1.7 Треды

В C++11 ввели модификатор *thread_local* показывающий что каждый тред будет иметь свою версию этой переменной, и её можно инициализировать, и она расположена в TLS[25] :

Листинг 1.6: C++11

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

thread_local int tmp=3;
```

[24] о том как возвращаются переменные интегральных типов из ф-ций, можно прочитать еще здесь [19, 1.6]
[25] Thread Local Storage

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

[26]

В исполняемом файле значение *tmp* будет именно в TLS.

Это удобно например для хранения глобальных переменных вроде *errno*, которая не может быть одна для всех тредов.

1.8 Ф-ция main()

Стандартное определение:

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

argc будет 1 при отсутствии аргументов, 2 — при одном аргументе, 3 — если два, и т.д..

- argv[0] — имя текущей запущенной программы.
- argv[1] — первый аргумент.
- argv[2] — второй аргумент.
- и т.д..

элементы в *argv* можно перечислять в цикле. К примеру, программа может принимать список файлов в командной строке (как это делает утилита UNIX *cat* и т.д.). Опции с дефисом в начале могут добавляться для отличия их от имен файлов.

В аргументах *main()*, *envp[]* может быть пропущено, но и *argc/argv[]*, и это корректно. Почему это корректно, можно прочитать в [19, 1.2.1].

Выражение return может быть пропущено начиная с C99 (1.6.1) (тогда ф-ция *main()* будет возвращать 0 [27]).

в CRT[28] возвращаемое значение ф-ции *main()* в итоге передается в ф-цию *exit()* либо в *ExitProcess()* в win32. Обычно это возвращаемый код ошибки, который можно проверять в шеллах, и т.д.. 0 обычно означает успех, хотя, разумеется, автор сам может определять (и переопределять) свои возвращаемые коды.

1.9 Разница между stdout/cout и stderr/cerr

stdout это то что выводится на консоль при помощи вызова *printf()* или *cout* в Си++. *stdout* это буферизированный вывод, так что, пользователь, обычно того не зная, видит вывод порциями. Бывает так что программа выдает что-то используя *printf()* или *cout* и тут же падает. Если это попадает в буфер, но буфер не успевает "сброситься" (flush) в консоль, то пользователь ничего не увидит. Это бывает неудобно. Таким образом, для вывода более важной информации, в том числе отладочной, удобнее использовать *stderr* или *cerr*.

stderr это не буферизированный вывод, и всё что попадает в этот поток при помощи `fprintf(stderr,...)` или *cerr*, появляется в консоли тут же.

Не следует также забывать, что из-за отсутствия буфера, вывод в *stderr* медленнее.

Чтобы направлять *stderr* в другой файл при запуске процесса, в командной строке (Windows/UNIX) можно указывать:

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

... это направит вывод *stderr* в заданный файл (потому что номер потока для вывода ошибок — 2).

1.10 Устаревшие особенности

1.10.1 register

Этим ключевым словом обозначались раньше переменные, которые компилятор должен был разместить в регистрах CPU (если это возможно), для более быстрого доступа к ним.

[26] Компилируется в GCC 4.8.1, но не в MSVC 2012
[27] это исключение из правил существует только для *main()*
[28] C runtime library

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

В наше время компиляторы уже достаточно развиты для того чтобы самостоятельно решать это, так что в использовании этого ключевого слова нет никакой необходимости. Впрочем, во время чтения старых исходных кодов, это помогает быстро увидеть наиболее используемые переменные в ф-ции.

Глава 2

Си

2.1 Работа с памятью Си

Есть наверное только два основных типа в памяти, предоставляемых программисту на Си.

- Память выделяемая в локальном стеке. Это локальные переменные, память выделенная при помощи alloca(). Обычно это очень быстро выделяемая память.
- Куча[1]. То что выделается при помощи malloc().

2.1.1 Локальный стек

Когда вы определяете что-то вроде `char a[1024]`, выделения памяти как такового не происходит, происходит просто перемещение указателя стека на 1024 байта назад [19, 1.2.3]. Это очень быстрая операция.

Освобождать эту память никак не надо, в конце работы ф-ции, это происходит автоматически, с возвратом указателя стека.

В качестве обратной стороны медали, вам нужно знать заранее, сколько места нужно выделить, а также, размер этого блока нельзя изменить, освободить и выделить заново его также нельзя.

Выделение локальных переменных происходит простым сдвигом указателя стека назад [**?**, 1.2.1]REBook]. При этом с выделенной областью памати ничего больше не происходит, в новых выделенных переменных будет содержаться то, что находилось в это время в этом месте в стеке, скорее всего, что-то от работы предыдущих ф-ций.

2.1.2 alloca()

Ф-ция alloca() выделяет блок памяти в локальном стеке точно также, отодвигая указатель стека [19, 1.2.4]. Блок памяти будет освобождена в конце ф-ции автоматически.

В стандарте C99(2.5.10), использовать alloca() уже не обязательно, там можно просто писать:

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

Это называется variable length array.

Впрочем, внутри, это работает так же как и alloca().

Критика: Линус Торвальдс против использования alloca() [18].

2.1.3 Выделение памяти в куче

Куча (heap) это какая-то часть памяти выделенная ОС процессу, где процесс может уже сам делить эту часть как хочет. После завершения процесса (в т.ч., некорректного), куча автоматически аннулируется и ОС не нужно разбирать по одному все выделенные процессом блоки.

Для работы с кучей есть стандартные библиотечные ф-ции malloc(), calloc(), realloc(), free(), а в Си++ — new/delete.

Очевидно, чтобы поддерживать информацию о выделенных блоках в куче, нужна масса связных друг с другом структур. Отсюда имеется вполне осязаемые накладные расходы (overhead). Вы можете выделить блок размером

[1] heap

8 байт, но еще как минимум 8 байт[2] будет задействованы для хранения информации о выделенном блоке [3]. В 64-битных ОС указатели занимают в два раза больше, так что информация о каждом блоке будет занимать как минимум 16 байт. В свете этого, чтобы эффективнее использовать память компьютера, блоки должны быть побольше, либо сама организация данных должна быть иная.

Использование кучи требует некоторой программистской дисциплины, без которой легко наделать ошибок. Возможно поэтому, считается что ЯП с RAII[4] как Си++ либо ЯП со сборщиками мусора (Python, Ruby) легче.

Одна из основных ошибок: утечки памяти

Память была выделена, но её забыли освободить через free(). Эта проблема довольно легко решается своей собственной надстройкой над ф-циями malloc()/free(). Пусть эта надстройка ведет учет выделенных блоков, а также, где и когда (и для чего) был выделен тот или иной блок.

Я сделал это в своей библиотеке octothorpe [5]. Макрос DMALLOC вызывает ф-цию dmalloc(), передавая ей имя файла, имя вызывающей ф-ции, номер строки, а также комментарий (имя блока). В конце работы программы, вызываем `dump_unfreed_blocks()` и он покажет список блоков, которые забыли освободить:

```
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
```

У каждого блока есть также номер. Это для того чтобы можно было установить брякпоинт по номеру выделяемого блока — тогда отладчик сработает в тот момент, когда этот блок будет выделяться и вы увидите, где и при каких условиях это происходит.

Писать в коде комментарии для каждого выделяемого блока памяти нудно, но очень полезно. Потом легко увидеть, под что была выделена память. Я впервые увидел эту идею в Oracle RDBMS. Помимо всего прочего, там еще и ведется статистика, под какие блоки было выделено больше памяти, её можно легко увидеть:

```
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
```

Подобная штука также присутствует и в ядре Windows, там это называется *tagging*.

При выделении памяти в ядре или драйвере, нужно указывать также 32-битный тег (обычно, четырехбуквенное сокращение, означающее подсистему Windows). Затем в отладчике можно увидеть статистику, под что выделено больше всего памяти:

```
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
```

[2] MSVC, 32-битная Windows, примерно то же самое и в Linux

[3] Это еще называют "метаданными", то есть, данные о данных

[4] Resource Acquisition Is Initialization

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

```
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
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
```

Конечно, можно возразить, что для этого нужно хранить еще больше информации о выделенных блоках, а еще и теги, названия блоков. И это еще сильнее замедляет работу программы. Конечно. Поэтому пусть это будет работать только в отладочных (debug) сборках, а в release-сборках, DMALLOC() становится обычной *пустой* функцией-переходником[6] для malloc(). Ну а в ядре Windows это вообще по умолчанию отключено, и нужно включать при помощи утилиты GFlags [7] Помимо всего прочего, подобное есть и в MSVC [8].

Одна из основных ошибок: разрушение кучи

Нетрудно выделить память под 4 байта, но по ошибке дописать туда пятый. Скорее всего, сразу это никак не проявится, но фактически это очень опасная мина замедленного действия, опасная, потому что ведет к трудновыявляемым ошибкам. Байт следующий за выделенным вами блоком может не использоваться вовсе, но также там уже может начинаться какая-то структура менеджера памяти, хранящая информацию о каком-то выделенном блоке, а может даже об этом самом. Если какую-то из таких структур сознательно разрушить, перезаписать, то тогда следующие вызовы malloc() или free() не смогут корректно работать. Иногда это проявляется в выводе ошибок вроде (в Windows):

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

Подобные ошибки эксплуатируются авторами эксплоитов: если знать что вы можете изменить структуры данных менеджера памяти нужным вам образом, вы можете добиться какого-то нужного вам поведения программы (это называется переполнение кучи (heap overflow) [9]).

Довольно распространенный метод борьбы с подобными ошибками: это просто дописывать "guard"-ы с обоих сторон блока, например, 4-байтного размера. Например, я сделал это в своем DMALLOC. При каждом вызове free(), проверяется целостность guard-ов (это могут быть просто какие-то фиксированные значения вроде 0x12345678), и если кто-то или что-то затерло один из них, можно тут же сообщить об этом.

Одна из основных ошибок: непроверка результата malloc()

При успешном выполнении, malloc() возвращает указатель на только что выделенный блок, который можно использовать, либо NULL если памяти не хватает. Конечно, в наше время дешевой памяти эта проблема становится редкой, тем не менее, если вы используете много памяти, думать об этом все же надо. Проверять возвращаемый указатель после каждого вызова malloc() неудобно, так что довольно популярный метод это писать свои функции-переходники с названием xmalloc(), xrealloc(), вызывающие malloc()/realloc(), но проверяющие их результат и падающие в случае ошибки.

Интересно упомянуть, как ведет себя xmalloc() в 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);
```

[6] thunk function
[7] http://msdn.microsoft.com/en-us/library/windows/hardware/ff549557(v=vs.85).aspx
[8] читайте больше о ф-циях _CrtSetDbgFlag и _CrtDumpMemoryLeaks
[9] https://en.wikipedia.org/wiki/Heap_overflow

```
                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;
}
```

[10]

Если malloc() не успешен, он пытается освободить какие-то уже выделенные (и не очень нужные) блоки при помощи try_to_free_routine(), а затем вызвать malloc() снова.

Помимо всего прочего, если определен XMALLOC_POISON, все байты в выделенном блоке заполняются 0xA5.

Это может помочь визуально, на глаз, увидеть когда вы, например, выделили память под структуру, а затем используете какое-то поле из нее до того как инициализировали.

Значение 0xA5A5A5A5 будет бросаться в глаза в отладчике, ну или просто если вы захотите где-то в дампе вывести его в шестнадцатеричной форме. В MSVC для этой же цели служит константа 0xbaadf00d.

И даже более того: после вызова free(), освобожденный блок может маркироваться уже какой-то другой константой, чтобы если кто-то захочет использовать что-то оттуда после освобождения блока, это также было видно, хотя бы визуально.

Некоторые константы от 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
```

[11]

Еще частые ошибки

Если не включить заголовочный файл stdlib.h, GCC считает возвращаемое значение неизвестной ф-ции malloc() за *int* и постоянно ругается на приведение типов.

С другой стороны, в Си++, результат malloc() все же нужно приводить к нужному вам типу:

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

Еще одна ошибка, которая может попортить нервов, это выделить один и тот же блок памяти в одном месте больше одного раза (предыдущие вызовы "теряются" из вида).

Еще методы борьбы

Однако, может оказаться так, что ошибки в программе у вас есть, а перекомпилировать её по каким-то причинам вы не можете. Тогда может помочь, например, valgrind.

2.1.4 Локальный стек или куча?

Конечно, в локальном стеке выделение памяти происходит намного быстрее.

[10] https://github.com/git/git/blob/master/wrapper.c
[11] https://en.wikipedia.org/wiki/Magic_number_(programming)

Например: в tracer [12] у меня есть дизассемблер[13] и эмулятор x86-процессора [14]. Когда я писал дизассемблер на Си (я делал это после того как длительное время писал на ЯП более высокого уровня — Python), я думал, что было бы неплохо, чтобы он сам выделял память под структуру, заполнял её и возвращал указатель на нее, а в случае ошибки дизассемблирования, возвращал бы NULL.

Эстетически, это неплохо смотрится, в стиле высокоуровневых ЯП, к тому же, такой код наверное легче читается. Однако, дизассемблер и эмулятор x86-процессора работают в цикле, огромное количество раз в секунду и эффективность здесь более чем важна. Так что, основной цикл у меня выглядит примерно так:

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

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

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

};
```

Затрат на выделение памяти под структуры, описывающие дизассемблированную инструкцию, нет вовсе. А иначе, нужно было бы на каждой итерации цикла вызывать malloc()/free(), каждая из которых, каждый раз, работала бы со структурами кучи, и т.д..

Как известно, у x86-инструкций может быть вплоть до трех операндов, так что, в моей структуре, помимо кода инструкции, есть также и информация о трех операндах. Конечно, можно было бы оформить её примерно так:

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

... а в случае отсутствия какого либо операнда, пусть там будет NULL. Тем не менее, это снова выделение памяти в куче.

Так что у меня сделано примерно так:

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

Такая структура занимает больше места в памяти. К тому же, трехоперандные инструкции очень редки в x86-коде, а здесь у меня пустой третий операнд хранится всегда. Однако, лишних манипуляций с памятью не происходит.

Ну а если уж так сильно хочется сэкономить на третьем операнде, то можно не хранить третий операнд вовсе: нетрудно вычислить размер структуры без одного операнда: `sizeof(disassembled_instruction) - sizeof(struct operand)` и скопировать её куда-то, где она должен храниться. Ведь никто не запрещает нам использовать (и хранить) не всю структуру а только её часть. А ф-ции работы с этой структурой могут не трогать в памяти третий операнд вовсе и, таким образом, ошибок не будет.

И даже более того: я специально сделал свой дизассемблер именно так, чтобы он мог принимать на вход не инициализированную структуру, и мог работать даже если там осталась информация от предыдущих вызовов.

Возможно, это уже слишком, но вы поняли идею.

Таким образом, если вы выделяете память под небольшие структуры заранее известного размера, или если скорость очень важна, то лучше подумать насчет выделения в локальном стеке.

[12] http://yurichev.com/tracer-ru.html
[13] https://github.com/dennis714/x86_disasm
[14] https://github.com/dennis714/bolt/blob/master/X86_emu.c

2.2 Строки в Си

Причина, почему формат строки в Си именно такой (оканчивающийся нулем) вероятно историческая. В [15] мы можем прочитать:

> 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.

В Си нет встроенных возможностей для удобной работы со строками, такими, какие имеются в ЯП более высокого уровня как конкатенация.

Часто жалуются на неудобную конкатенацию строк (то есть, склеивание) в Си при помощи функции strcat(). Также, многих раздражает sprintf(), под который нельзя толком заранее предсказать, сколько нужно выделять памяти.

Копирование строк при помощи strcpy() также неудобно — нужно думать, сколько же выделить байт под буфер. Помимо всего прочего, неудобная работа со строками в Си, это источник огромного количества уязвимостей в ПО, связанных с переполнениями буфера [19, 1.14.2].

Прежде всего, нужно задать себе вопрос, какие операции со строками нам нужны. Конкатенация (склеивание) нужна чтобы 1) выдавать в лог сообщения; 2) конструировать строки и затем передавать (или записывать) их куда-то.

Для 1) можно использовать потоки (streams) — не конструируя строку, выдавать её по порциям, например:

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

Подобное заменяется в Си++ выводом в *ostream*:

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

Так быстрее и меньше требуется памяти для конструирования строк.

Кстати, ошибкой является писать так:

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

Для неспешного вывода в лог небольшого кол-ва сообщений это нормально, но если таких сообщений очень много, то будут накладные расходы на их конкатенацию.

Но все же строки иногда конструировать надо.

Есть какие-то библиотеки для этого. К примеру, в Glib[15] есть gstring.h[16]/ gstring.c[17].

А в исходниках git можно найти strbuf.h[18]/ strbuf.c[19]. Собственно, подобные Си-библиотеки очень похожи: они обеспечивают структуру данных, в которой есть некоторый буфер для строки, текущий размер буфера и текущий размер строки в буфере. При помощи отдельных функций, можно добавлять новые строки или символы в буфер, который, в свою очередь, будет автоматически увеличиваться или даже уменьшаться.

В *strbuf.c* из git есть в том числе и ф-ция *strbuf_addf()*, работающая как *sprintf()*, но добавляющая строку-результат в буфер.

Так программист освобождается от головной боли связанной с выделением памяти. При работе с этими библиотеками, практически невозможна ситуация переполнения буфера, если только не работать со структурой данных самостоятельно.

Типичная последовательность работы с такими библиотеками, выглядит так:

- Инициализация структуры strbuf или GString.
- Добавление строк и/или символов.

[15] https://developer.gnome.org/glib/
[16] https://github.com/GNOME/glib/blob/master/glib/gstring.h
[17] https://github.com/GNOME/glib/blob/master/glib/gstring.c
[18] https://github.com/git/git/blob/master/strbuf.h
[19] https://github.com/git/git/blob/master/strbuf.c

- Имеем сконструированную строку.
- Модифицируем её если нужно.
- Используем её как обычную Си-строку, записываем куда-то в файл, передаем по сети, и т.д..
- Освобождаем структуру.

Кстати, конструирование строк чем-то напоминает Buffer[20], ByteBuffer[21] и CharBuffer[22] в Java.

2.2.1 Хранение длины строки

Всегда хранить длину строки — это было принято в реализациях ЯП Pascal. Не смотря на исходы святых войн[23] между приверженцами Си и Pascal, все же, почти все библиотеки для хранения строк и работы с ними, хранят также и текущую длину — просто потому что удобства от этого перевешивают необходимость пересчитывать это значение после каждой модификации.

Например, *strlen()* [24] больше не нужен вообще, длина строки известна всегда. Конкатенация строк работает намного быстрее, потому что не нужно вычислять длину первой строки. Ф-ция сравнения строк в самом начале может сравнить длины строк и если они не равны, тут же вернуть *false*, не начиная сравнивание символов в строках.

В всетевых библиотеках Oracle RDBMS, в функции работы со строками, зачастую передается строка и, отдельным аргументом, её длина [25]. Это не очень эстетично, это выглядит избыточно, зато очень удобно. Например, у нас есть некоторая ф-ция, которой нужно в начале узнать, какую строку ей передали:

```
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");
};
```

А вот если бы эта ф-ция имела длину входной строки, её можно было бы переписать так:

```
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)
```

[20] http://docs.oracle.com/javase/7/docs/api/java/nio/Buffer.html
[21] http://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html
[22] http://docs.oracle.com/javase/7/docs/api/java/nio/CharBuffer.html
[23] holy wars
[24] подсчёт длины строки
[25] http://blog.yurichev.com/node/64

```
                                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");
};
```

Конечно, с эстетической точки зрения, код выглядит ужасно. Тем не менее, мы здорово сократили количество необходимых сравнений строк! Вероятно, для тех ситуаций, когда нужно как можно быстрее обрабатывать текстовые строки, такой подход может улучшить ситуацию.

2.2.2 Возврат строки

Если некая ф-ция должна вернуть строку, имеются такие возможности:

- 1: Возврат строки-константы, это самое простое и быстрое.
- 2: Возврат строки через глобальный массив символов. Недостаток: массив один и каждый вызов ф-ции перезаписывает его содержимое.
- 3: Возврат строки через буфер, заданный в аргументах ф-ции. Недостаток: нужно также передавать и длину буфера, и вообще его длину нельзя заранее правильно расчитать.
- 4: Выделяем буфер нужного размера сами, записываем туда строку, возвращаем указатель. Недостаток: тратятся ресурсы на выделение памяти.
- 5: Записываем строку в уже рассмотренный `strbuf` или `GString` или иную другую структуру, указатель на которую был передан в аргументах.

2.2.3 1: Возврат строки-константы

Первый вариант очень прост. Например:

```
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!";
        };
};
```

Можно даже еще проще:

```
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: Через глобальный массив символов

Так делает стандартная ф-ция `asctime()`. Следует помнить, что нужно использовать возвращенную строку перед каждым следующим вызовом `asctime()`.

Например, это правильно:

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

А это нет:

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

... ведь указатели `date1` и `date2` будут указывать на одно и то же место, и вывод `printf()` будет одинаковым.

В git в *hex.c*[26] можно найти такое:

```
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;
}
```

Строка возвращается фактически через глобальную переменную, определение её как `static` внутри ф-ции просто напросто обеспечивает доступ к ней только из этой ф-ции. Но вот недостаток: после вызова *sha1_to_hex()* вы не можете вызвать её повторно для получения второй строки до тех пор, пока не используете как-то первую, ведь она затрется. Для того чтобы решить эту проблему здесь, по видимому, сделали сразу 4 буфера и каждый раз строка возвращается в следующем. Но имейте ввиду — так можно делать если только вы уверены в том что вы делаете,

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

это код на уровне "грязного хака". Если вы вызовете эту ф-цию 5 раз и вам нужно будет использовать как-то строку полученную при первом вызове, это может привести к трудновыявляемой ошибке.

Кстати, обратите также внимание на то что переменная *bufno* не инициализируется, потому что используются только 2 младших её бита, к тому же, не важно, какое значение переменная будет содержать в самом начале.

2.2.5 Стандартные ф-ции в Си для работы со строками

Некоторые ф-ции, например, getcwd() не только заполняют буфер, но и возвращают указатель на него. Это для того чтобы можно было писать что-то вроде:

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

... вместо:

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

strstr() и memmem()

strstr() применяется для поиска строки в другой строке, либо чтобы узнать, есть ли там такая строка вообще.

memmem() можно применять с этими же целями, но для поиска по буферу, в котором могут быть нули, либо по части строки.

strchr() и memchr()

strchr() применяется для поиска символа в строке, либо чтобы узнать, есть ли там такой символ вообще.

memchr() можно применять с этими же целями, но для поиска по части строки.

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

Ф-ции atoi()/atof() не могут сигнализировать об ошибке, а strtod()/strtof() , делая то же самое — могут.

scanf(), fscanf(), sscanf()

Извечный спор, что лучше, текстовые файлы или бинарные. Бинарные файлы быстрее и проще обрабатывать, зато текстовые легче просматривать и редактировать в любом текстовом редакторе, к тому же, в UNIX имеется огромный арсенал утилит для обработки текстов и строк. Но текстовые файлы нужно парсить.

Ф-ции scanf() [6, 7.19.6/2] конечно же, регулярные выражения не поддерживают, однако при их помощи некоторые простые последовательности строк можно парсить.

Пример #1 Генерируемый ядром Linux файл `/proc/meminfo`, начинается примерно так:

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

Предположим, нам нужно узнать первое и третье число, игнорируя второе и остальные. Так это можно сделать:

```
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);
};
```

Строка формата расходится на три строки, в реальности это одна: (1.2.2). N.B. Перевод строки задается как \n.

* в модификаторе scanf-строки указывает что число будет прочитано, но никуда записано не будет. Таким образом, это поле игнорируется. scanf()-функции возвращают кол-во не прочитанных полей (здесь их будет 3) а кол-во записанных полей (2).

Пример #2 Имеется текстовый файл с парами в каждой строке (ключ-значение):

```
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
```

Нужно просто читать оба поля:

```
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[^=] — это отдаленно напоминает регулярные выражения. Означает, читать 16 любых символов, кроме знака "равно" (=). Затем, мы указываем scanf()-у, что далее должен быть этот самый знак (=). Затем пусть он читает 60 любых символов, кроме символа перевода строки. В конце читаем символ перевода строки.

Это работает, и поля ограничены длиной 16 и 60 символов. Поэтому на 5-й строке предсказуемо происходит ошибка, ведь там длина парамера (первое поле) длиннее.

Так можно парсить несложные форматы, даже CSV[27].

Однако, нельзя забывать о том что scanf()-функции не способны прочитать пустую строку там где задается модификатор %s. Поэтому, этим методом невозможно парсить файл с ключами-значениями, где есть отсутствующие ключи или значения.

Засада #1 Если использовать %d в строке формата, scanf() подразумевает что это 32-битный int и на x86 и на x64 процессорах.

Частой ошибкой является писать нечто подобное:

```
char a[10];

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

[27] Comma-separated values

Символы (или байты) лежат "в притык" друг к другу. Когда scanf() будет обрабатывать первое значение, он будет считать его за 32-битный `int`, и "затрет" остальные три, рядом лежащие. И так далее.

strspn(), strcspn()

`strspn()` часто применяется для того чтобы удостовериться, что некая строка полностью состоит из нужных символов:

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

Либо для того чтобы пропустить начало строки:

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

`strcspn()` это обратная ф-ция, её можно использовать для пропуска всех символов в начале строки, не попадающих под множество символов:

```
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() и strpbrk()

Обе ф-ции служат для разбиения строки на подстроки, отделенные друг от друга разделительными символами [28]. Только strtok() модифицирует исходную строку (и таким образом, получаемые подстроки сразу можно использовать как отдельные Си-строки), а strpbrk() нет, он только возвращает указатель на следующую подстроку.

2.2.6 Unicode

Unicode в наше время это важно. Наиболее популярные способы его применения это:

- UTF-8 Популярно в UNIX-системах. Сильное приемущество: можно продолжать пользоваться многими стандартными (и не только) ф-циями для обработки строк.
- UTF-16 Используется в Windows API.

UTF-16

Под каждый символ отводят 16-битный тип *wchar_t*.

Для объявления строк с таким типом, используется макрос *L*:

```
L"hello world"
```

Для работы с wchar_t вместо char, имеется целый класс функций-двойников с символом w в названии, например: fwprintf(), wcscmp(), wcslen(), iswalpha().

Windows В Windows, если некто хочет писать программу сразу в двух версиях, с использованием Unicode и без, для этого есть тип `tchar`, в зависимости от объявленной переменной препроцессора `UNICODE`, он будет либо `char` либо `wchar_t` [29]. Для этого же имеется макрос `_T(...)`:

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

В зависимости от выставленной переменной препроцессора `UNICODE`, она будет определена как `char` или `wchar_t`.

В заголовочном файле `tchar.h` есть масса ф-ций, меняющих свое поведение в зависимости от этой переменной.

[28] delimiter

[29] Одновременные сборки с Unicode и без были популярна во времена популярности как Windows NT/2000/XP так и Windows 95/98/ME. Вторая линейка плохо поддерживала Unicode

2.2.7 Списки строк

Самый простой список строк, это просто набор строк оканчивающийся нулем. Например, в Windows API, в библиотеке Common Dialogs так [30] передаются список допустимых расширений файлов для диалогового окна:

```
// 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 Ваши собственные структуры данных в Си

2.3.1 Списки в Си

Списки это связный набор элементов. Односвязный список — это когда у каждого элемента есть ссылка на следующий. Двусвязный список — когда у элемента есть ссылки на следующий и на предыдущий.

У списков есть серьезное преимущество перед массивами: в список легко добавлять элемент в произвольное место, так и удалять. В качестве недостатков: тратится много памяти для поддержания самих структур списка, а также нет возможности индексировать список как массив.

Односвязный список

Самый легкий для реализации. В структуре предназначенной для связывания в список, достаточно добавить где-то ссылку на следующий элемент, обычно это поле называется `next`:

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

`NULL` в `next` означает что этот элемент является последним в списке.

Операция прохода по такому списку становится очень простой:

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

Для вставки нового элемента, нужно вначале найти последний элемент:

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

... а затем, создав новую структуру, записать указатель на нее в `next`:

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

calloc() отличается от malloc() тем что обнуляет всё выделенное место, а значит в поле `next` нового элемента сразу будет NULL [31].

Поиск нужного элемента это просто проход по всему списку и сравнение каждого элемента с искомым, до тех пор, пока не найдется то что нужно.

Удаление элемента: найти предыдущий элемент и следующий, у предыдущего в `next` установить указатель на следующий элемент, затем освободить блок памяти выделенный для текущего элемента.

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

[31] Об "инициализации" структур, читайте также здесь: (2.4.1).

Самый первый элемент списка называется "list head". Структуру самого первого элемента можно определить как локальную или глобальную переменную. Но тогда удалять первый элемент списка будет неудобно. А с другой стороны, можно определить указатель на первый элемент списка, тогда будет проще этому указателю присвоить другой элемент, который будет первым.

Двусвязный список

Это почти то же самое, только, помимо указателя на следующий элемент, хранится еще и указатель на предыдущий. Если элемент первый, то указатель на предыдущий элемент может быть NULL, либо он может указывать сам на себя (кому как удобнее).

Работая с двусвязным списком, легче находить предыдущие элементы, например, когда нужно удалить какой-то элемент. А также можно перебирать элементы с конца списка до начала. Но памяти на это тратится немного больше.

Нередко, двусвязный список также является кольцевым, т.е., первый и последний элементы указывают друг на друга. Например, так сделано в std::list в C++ STL [19, 2.4.2]. Это значительно ускоряет поиск последнего элемента (не нужно перебирать все элементы).

Windows API

Здесь, да и много где в ядре Windows, применяются две примитивные структуры:

```
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;
```

Эти структуры нельзя назвать самостоятельными, они скорее предназначены для встраивания в другие структуры. Например, нам нужно объединить в список структуру описывающую цвет:

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

Теперь в вашей структуре есть также и ссылка на предыдущий элемент и на следующий. Для работы со структурами использующие эти списки, в Windows есть набор ф-ций [32].

Linux

В ядре Linux работа с простыми двусвязными списками, описывается в файле /include/linux/list.h [33].

Там это много где используется, в ядре версии 3.12 по крайней мере 2900 упоминаний "struct list_head".

Glib

Напрашивается мысль, а нельзя ли выделить отдельную структуру для элемента списка, и не встраивать лишних полей в свои структуры? Можно, например, так сделано в glist.h [34] в Glib:

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

[32] http://msdn.microsoft.com/en-us/library/windows/hardware/ff563802(v=vs.85).aspx

[33] http://lxr.free-electrons.com/source/include/linux/list.h

[34] https://github.com/GNOME/glib/blob/master/glib/glist.h https://developer.gnome.org/glib/2.37/glib-Doubly-Linked-Lists.html

data может указывать на какой угодно объект, на любую существующую структуру, в которой вы ничего не хотите менять, это также называется "opaque pointer". Конечно, с эстетической точки зрения, это лучше. Но нельзя забывать, что тогда на каждый элемент вашего списка, будет приходится уже два выделенных блока памяти + еще затраты на поддержания самих блоков памяти в куче(2.1.3).

Таким образом, подобное решение оправдано там, где экономия памяти менее важна.

2.3.2 Бинарные деревья в Си

Бинарные деревья — одна из важнейших структур данных в компьютерных науках. Чаще всего они используются для хранения пар "ключ-значение". Это то что в Си++ STL[35] реализовано в std::map.

Упрощенно говоря, по сравнению со списками, выборка у деревьев происходит намного быстрее. С другой стороны, добавление элемента в дерево может происходить медленнее.

В стандартных библиотеках Си, нет работы с деревьями, но кое-что есть в POSIX[36] (tsearch(), twalk(), tfind(), tdelete()) [37].

Это семейство ф-ций активно используется в Bash 4.2, BIND 9.9.1, GCC — там можно посмотреть, как это использовать.

В Glib имеется также свои ф-ции для работы с деревьями, определенные в gtree.h [38].

Множество (std::set в Си++ STL) можно реализовать так же просто при помощи бинарных деревьев, достаточно просто не хранить значение, а хранить только ключ.

2.3.3 Еще кое что

Структуры данных описывающие какие-либо коллекции, могут также содержать и указатели на ф-ции для работы с элементами, например, ф-ции сравнения, копирования, и т.д..

К примеру в GTree в Glib:

Листинг 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;
};
```

Задав ф-цию сравнения ключей, значений, а также ф-цию освобождения памяти (в g_tree_new_full()), ф-ции работы с деревьями в Glib смогут самостоятельно сравнивать два дерева, либо освобождать все структуры связанные с деревом.

2.4 Объектно-ориентированное программирование в Си

Как известно, в Си нет поддержки ООП, она есть в Си++, тем не менее, в "чистом" Си вполне можно программировать в стиле ООП.

ООП, коротко говоря, это явное разделение на объекты и методы. В Си структуры легко могут представляться объектами, а обычные ф-ции — методами.

2.4.1 Инициализация структур

В Си++ у классов имеются конструкторы. Если вам нужно каким-то особенным образом инициализировать структуру, вам и в Си придется делать подобную ф-цию. Но если структура простая, то её можно инициализировать при помощи calloc() [39] или bzero()(2.5.3).

[35] Standard Template Library
[36] Portable Operating System Interface
[37] http://pubs.opengroup.org/onlinepubs/009696799/functions/tsearch.html
[38] https://github.com/GNOME/glib/blob/master/glib/gtree.h
[39] Это тоже самое что и malloc() + заполнение выделенной памяти нулями

Все int-переменные становятся нулями. Нулевое значение bool в C99(2.5.10) и Си++ это false, так же как и BOOL в Windows API. Все указатели становятся NULL. И даже вещественный ноль представляемый в формате IEEE 754 это также все ноли во всех битах.

Если в структуре присутствуют указатели на другие структуры, то NULL может означать "отсутствие объекта".

Помимо всего прочего, неинициализированные глобальные переменные также обнуляются [6, 6.7.8.10].

Инициализация это важный момент. Очень трудно искать ошибки связанные с работой с неинициализированными переменными, а в случае со структурами, компилятор не предупредит, если вы используете переменную оттуда без инициализации.

2.4.2 Деинициализация структур

Если в структуре есть ссылки на другие структуры, то их нужно освобождать. В простом случае, обычным вызовом free(). Кстати, вот почему free() может принимать на вход NULL, это чтобы можно было просто писать `free(s->field)` вместо `if (s->field) free(s->field)`, так короче.

2.4.3 Копирование структур

Если структура простая, то её можно копировать обычным побайтовым копированием memcpy()(2.5.2). Если в такой манере скопировать структуру, в которой есть указатели на другие структуры, то это будет называться "shallow copy"[40]. И напротив, *deep copy* — это копирование структуры плюс всех связанных с ней структур (это дольше).

Вот почему может быть удобнее хранить строку в структуре как обычный массив символов фиксированной длины. Такого, например, очень много в Windows API. Такую структуру легко скопировать, её хранение требует меньших накладных расходов [41] в куче. Но с другой стороны, придется согласиться с ограничением на длину строки.

Помимо всего прочего, структуру можно копировать просто так: `s1=s2` — в итоге генерируется код, копирующий все поля по порядку. И это наверное легче читается чем вызов memcpy() на этом же месте.

2.4.4 Инкапсуляция

Си++ предлагает инкапсуляцию (сокрытие информации). Например, вы не можете написать программу модифицирующую защищенное поле в классе, это защита на стадии компиляции [19, 1.7.3].

В Си этого нет, поэтому тут нужно больше дисциплины.

Впрочем, можно попытаться "защитить" структуру "от посторонних глаз". Например, в Glib, имеется библиотека для работы с деревьями. В заголовочном файле gtree.h[42] нет описания самой структуры (она есть только в gtree.c [43]), а есть только forward declaration(1.2.1). Так разработчики Glib могут понадеятся что пользователи GTree постараются не пользоваться отдельными полями в структуре напрямую.

Но у такого метода есть и обратная сторона: могут быть крохотные однострочные ф-ции вроде "вернуть длину строки" в strbuf(2.2), например:

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

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

Если компилятору на стадии компиляции доступно и описание структуры и тело ф-ции, то в каком-то месте, вместо вызова strbuf_get_len() он может сделать эту ф-цию как inline-овую, вставить её тело прямо на том же место и сэкономить на вызове другой ф-ции. Но если эта информация компилятору недоступна, то он оставит вызов strbuf_get_len() как есть.

То же самое касается поля `buf` в структуре `strbuf`. Компилятор может генерировать куда более эффективный код, если этот сгенерированный машинный код сможет обращаться к полям структур на прямую, а не вызывать суррогатные функции-"методы".

[40] https://en.wikipedia.org/wiki/Object_copy
[41] overhead
[42] https://github.com/GNOME/glib/blob/master/glib/gtree.h
[43] https://github.com/GNOME/glib/blob/master/glib/gtree.c

2.5 Стандартные библиотеки Си

2.5.1 assert

Как известно, этот макрос часто используется для валидации [44] заданных значений. Например, если ваша ф-ция работает с датой, вы, вероятно, захотите написать в её начале что-то вроде: *assert (month>=1 && month<=12)*.

Вот то о чем нужно помнить: стандартный макрос assert() доступен только в отладочных (debug) сборках. В release, где объявлена переменная NDEBUG, все выражения как бы исчезают. Поэтому писать, например, *assert(f=malloc(...))* неверно. Впрочем, вы возможно захотите использовать что-то вроде *assert(object->get_something()==123)*.

В макросах *assert* можно также указывать небольшие сообщения об ошибках: вы увидите их если выражение "не сойдется". Например, в исходниках LLVM[45] можно встретить такое:

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

Текстовая строка имеет тип *const char**, и она никогда не NULL. Таким образом, можно дописать к любому выражению *... && true* не меняя его смысл.

Макрос assert() можно также вводить и для документации кода.

Например:

Листинг 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;
}
```

Глядя на этот код, можно очень быстро увидеть "легальные" значения переменных n и r.

assert-ы также называют "active comments" [11].

2.5.2 UNIX time

В UNIX-среде очень популярно представление даты и времени в формате UNIX time. Это просто 32-битное число, отсчитывающее количество прошедших секунд с 1-го января 1970-го года.

В качестве положительных сторон: 1) очень легко хранить это 32-битное число; 2) очень легко вычислять разницу дат; 3) невозможно закодировать неверные даты и время, такие как 32-е января, 29-е февраля невысокосных годов, 25 часов 62 минуты.

В качестве отрицательных сторон: 1) нельзя закодировать дату до 1970-го года.

В наше время, если использовать формат UNIX time, тем не менее, следует помнить что "срок его действия" истечет в 2038-м году, именно тогда 32-битное число переполнится, то есть, пройдет 2^{32} секунд с 1970-го года. Так что, для этого следует использовать 64-битное значение, т.е., `time64`.

2.5.3 memcpy()

Поначалу трудно запомнить порядок аргументов в ф-циях memcpy(), strcpy(). Чтобы было легче, можно представлять знак "=" ("равно") между аргументами.

2.5.4 bzero() и memset()

`bzero()` это ф-ция просто обнуляющая блок памяти. Для этого обычно используют `memset()`. Но у `memset()` есть неприятная особенность, легко перепутать второй и третий аргументы местами, и компилятор промолчит, потому что байт для заполнения всего блока задается как *int*.

[44] используется также такой термин как "инвариант" и "sanitization" в англ.яз.

[45] http://llvm.org/

К тому же, имя ф-ции bzero() легче читается.

С другой стороны, её нет в стандарте Си, но она есть в POSIX.

А в Windows API для этих же целей применяется ZeroMemory() [46] .

2.5.5 printf()

Ограничить длину выводимой строки:

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

Иногда бывает необходимость делать отступы в N пробелов или символов табуляции перед каждым выводимым сообщением:

```
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

Вывод типа size_t в MSVC: "%Iu", ssize_t: "%Id".

В GCC, соответственно: "%zu", "%zd".

Вывод DWORD в MinGW: "%lu" или "%lx".

Засада #1

Этот код:

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

... выдает FFFFFF80 потому что *char* по умолчанию знаковый (как в GCC так и в MSVC). При передаче в *printf()* как аргумента, он знаково расширяется до *int* и мы видим то что видим. Правильно так:

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

Свои собственные модификаторы в printf()

Часто можно испытать раздражение, когда было бы логично передать в printf(), скажем, структуру описывающее комплексное число, или цвет закодированный в структуре из трех чисел типа int.

Эту проблему в Си++ решают определением ф-ции operator« в ostream для своего типа, либо введением метода с названием ToString() (3.4).

В printk() (printf-подобная ф-ция в ядре Linux) имеются дополнительные модификаторы [47], такие как %pM (Mac-адрес), %pI4 (IPv4-адрес), %pUb (UUID[48]/GUID[49]).

В GNU Multiple Precision Arithmetic Library есть ф-ция gmp_printf() [50] имеющая нестандартные модификаторы нужные для вывода BigInt-чисел.

В ОС Plan9, и в исходниках компилятора Go, можно найти ф-цию fmtinstall(), для объявления нового модификатора printf-строки, например:

Листинг 2.3: go\src\cmd\5c\list.c

```
void
listinit(void)
{
```

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

[47] http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/printk-formats.txt

[48] Universally unique identifier

[49] Globally Unique Identifier

[50] http://gmplib.org/manual/Formatted-Output-Strings.html

```
        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);
        } 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)

Ф-ция Pconv() будет вызвана если в строке формата будет встречен %P. Затем она копирует созданную строку при помощи fmtstrcpy(). Кстати, эта ф-ция и сама использует другие объявленные модификаторы, такие как %A, %D, и т.д..

В glibc есть нестандартное расширение [51], позволяющее объявлять свои модификаторы, но это *deprecated*.

Попробуем определить свои собственные модификаторы для Mac-адреса и для вывода байта в бинарном виде:

```
#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]);

        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;
};
```

[52]

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

[52] Основа для примера взята отсюда: http://codingrelic.geekhold.com/2008/12/printf-acular.html

Это компилируется с предупреждениями:

```
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 умеет следить за соответствиями модификаторов в printf-строке и аргументами в вызове printf(), но здесь ему встречаются незнакомые модификаторы, о чем он предупреждает.

Тем не менее, наша программа работает:

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

2.5.6 atexit()

При помощи `atexit()` можно добавить ф-цию, автоматически вызываемую перед выходом из вашей программы. Кстати, программы на Си++ именно при помощи `atexit()` добавляют деструкторы глобальных объектов.

Можно попробовать:

```
#include <string>

std::string s="test";

int main()
{
};
```

В листинге на ассемблере найдем конструктор этого глобального объекта:

Листинг 2.4: MSVC 2010

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

Конструктор, конструируя, также регистрирует деструктор объекта в `atexit()`.

2.5.7 bsearch(), lfind()

Удобные ф-ции для поиска чего-либо где-либо.

Разница между ними только в том что lfind() просто ищет заданное, а bsearch() требует отсортированный массив данных, но зато может искать быстрее [53] [54].

К примеру, поиск строки в массиве указателей на строки:

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

Свой собственный stricmp() нужен потому что lfind() будет передавать в него указатель на искомую строку, а также на место в массиве где записан указатель на строку, но не сама строка. Если бы это был массив строк фиксированной длины, тогда можно было бы воспользоваться стандартным stricmp().

Кстати, точно также ф-ция сравнения задается и для qsort().

lfind() возвращает указатель на место в массиве где ф-ция `my_stricmp()` сработала выдав 0. Далее вычисляем разницу между адресом этого места и адресом начала самого массива. Учитывая арифметику указателей(1.3.7), в итоге получается кол-во элементов между этими адресами.

Реализуя ф-цию сравнивания, можно искать строку в каком угодно массиве. Пример из OpenWatcom:

Листинг 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 ) {
```

[53] методом половинного деления, и т.д.

[54] разница еще также в том что bsearch() есть в стандарте [6], а lfind() нет, он есть только в POSIX и в MSVC, но эти функции достаточно просты, чтобы их реализовать самому

```
    KeyWord     *match;

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

Здесь имеется отсортированный по первому полю массив структур *ReservedWords*, выглядящий так:

Листинг 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() ищет строку сравнивая её со строкой в первом поле структуры. Здесь можно применить именно bsearch(), потому что массив уже отсортированный. Иначе пришлось бы использовать lfind(). Вероятно, несортированный массив можно вначале отсортировать при помощи qsort(), а затем уже использовать bsearch(), если вам нравится такая идея.

Точно так же можно искать что угодно, где угодно. Пример из BIND 9.9.1 [55]:

Листинг 2.7: backtrace.c

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

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

```
        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;
```

Таким образом, можно обойтись без того чтобы писать каждый раз цикл for() для перебирания элементов, и т.д..

2.5.8 setjmp(), longjmp()

В каком-то смысле, это реализация исключений в Си.

jmp_buf это просто структура содержащая в себе набор регистров, но самые важные: это адрес текущей инструкции и адрес указателя стека.

Всё что делает setjmp() это просто записывает текущие регистры процессора в эту структуру. А всё что делает longjmp() это восстанавливает состояние регистров.

Это часто используется для возврата из каких-то глубоких мест наружу, обычно, в случае ошибок. Собственно, как и исключения в Си++.

Например, в Oracle RDBMS, когда происходит некая ошибка, и пользователь видит код и сообщение об ошибке, в реальности, там срабатывает longjmp() откуда-то из глубины. Для того же это используется и в Bash.

Этот механизм даже немного гибче чем исключения в других ЯП — нет никаких проблем устанавливать точки возврата в каких угодно местах программы и затем возвращаться туда по мере необходимости.

Пофантазируя, можно даже сказать что longjmp() это такой супер-мега-goto, обходящий блоки, ф-ции, и восстанавливающий состояние стека.

Однако, нельзя забывать, что всё что восстанавливает longjmp() это регистры процессора. Выделенная память остается выделенной, никаких деструкторов, как в Си++, вызвано не будет. Никакого RAII здесь нет. С другой стороны, так как часть стека просто аннулируется, память выделенная при помощи alloca()(2.1.1) будет также аннулирована.

2.5.9 stdarg.h

Тут ф-ции для обработки переменного количества аргументов. Как минимум ф-ции семейства printf() и scanf() такие.

Засада #1

Переменную типа va_list можно использовать только один раз, если нужно больше, их нужно копировать:

```
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() и rand()

ГПСЧ[56] из стандартной библиотеки очень низкокачественный, к тому же генерирует только числа в пределах 0..32767. Его лучше не использовать.

[56] Генератор псевдослучайных чисел

2.6 Стандарт Си C99

Текст стандарта: [6].

Этот стандарт поддерживается в GCC, Clang, Intel C++, но не в MSVC, и не ясно, будет ли он там поддерживаться вообще.

Чтобы включить его поддержку в GCC, нужно добавить ключ компилятора *-std=c99*.

Глава 3

Си++

3.1 Кодирование имен

В Си к именам прибавляется спереди символ подчеркивания, и ваша ф-ция с именем `function` может иметь имя в объектом файле `_function`.

В Си++ возможна перегрузка операторов, следовательно, несколько ф-ций могут иметь одно и то же имя, но разные типы. С другой стороны, загрузчик ОС и линкер могут работать только с обычными именами ф-ций (или символов) и ничего про Си++ не знают. Следовательно, имеется потребность кодировать имя функции, типы аргументов, тип возвращаемого значения, возможно имя класса, в одну строку.

К примеру, конструктор класса *box* объявленный следующим образом:

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

... в соглашениях MSVC, будет иметь следующее внутреннее имя: `??0box@@QAE@HHHH@Z` — например, четыре символа *H* означают четыре подряд идущих аргумента типа *int*.

Это называется *name mangling*.

Вот зачем в заголовочных файлах нужна директива `extern "C"`:

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

   void foo(int a, int b);

#ifdef  __cplusplus
}
#endif
```

Это означает что `foo()` написана на Си, скомпилирована как ф-ция Си и будет иметь имя в объектных файлах `_foo`.

Если этот заголовочный файл подключить к проекту на Си++, то компилятор будет знать, что имя ф-ции `_foo`. Без этого объявления, компилятор будет искать в объектных файлах ф-цию с внутренним именем `?foo@@YAXHH@Z`.

Таким образом, эта директива необходима для подключения Си-библиотек к Си++-проектам.

А *ifdef* делает эту директиву видимой только в Си++.

Еще о *name mangling*: [19, 1.17].

3.2 Объявления в Си++

3.2.1 C++11: auto

Пользуясь STL, иногда надоедает каждый раз определять тип итератора вроде:

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

Тип *it* вполне можно получить из `list.begin()`, поэтому, в начиная со стандарта C++11 можно использовать *auto*:

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

3.3 Элементы языка Си++

3.3.1 references

Это то же что и указатели (1.3.7), но "обезопасенные" (safe), потому что работая с ними, труднее сделать ошибку [8, 8.3.2].

Например, reference всегда должен указывать объект того же типа и не может быть NULL [2, 8.6]. Более того, reference-ы нельзя менять, нельзя его заставить указывать на другой объект (reseat) [2, 8.5].

В [19, 1.7.1] продемонстрировано, что на уровне x86-кода, это одно и то же.

Как и указатели, *reference* также можно возвращать из ф-ций, например:

```
#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;
};
```

Это выведет предсказуемое:

```
1001
1002
1003
1004
```

Если посмотреть что сгенерировалось на языке ассемблера, то можно увидеть что `use_count()` просто возвращает указатель на `uc`, а в `main()` происходит инкремент значения по этому указателю.

3.4 Ввод/вывод

Часто есть необходимость выводить в *ostream* целые структуры, а каждый раз выводить их по одному полю это неудобно. Иногда это решается добавлением метода `ToString()` в класс. Другое решение это сделать свободную ф-цию для вывода вроде:

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

    return out;
};
```

Теперь можно отправлять объекты прямо в *ostream*:

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

А для того чтобы ф-ция вывода могла обращаться к любым полям класса, её можно сделать *friend*. Впрочем, имеется также такая точка зрения, что подобные ф-ции не следует делать *friend* для сохранения энкапсуляции [12, Item 23 Prefer non-member non-friend functions to member functions].

3.5 Темплейты

Темплейты нужны обычно для того чтобы сделать класс универсальным для нескольких типов данных. К примеру, `std::string` в реальности это `std::basic_string<char>`,
а `std::wstring` это `std::basic_string<wchar_t>`.

Нередко подобное делают и для типов *float/double/complex* и даже *int*. Некий математический алгоритм может быть описан один раз, но скомпилирован сразу в нескольких версиях, для работы со всеми этими типами данных.

Таким образом, можно описывать алгоритмы только один раз, но работать они будут для разных типов.

Простейшие примеры это ф-ции *max, min, swap* , работающие для любых типов, переменные которых можно сравнивать и присваивать. Потом, вы можете написать свою реализацию BigInt, и если там присутствует оператор сравнения двух объектов (`operator<`), то написанные раннее *max/min* будут работать и для нового класса.

Вот почему списки и прочие контейнеры в STL это именно темплейты: они как бы "встраивают" в ваш класс возможность объеденять в списки, и т.д..

3.6 Standard Template Library

Внутренности `std::string`, `std::list`, `std::map` и `std::set` описаны в [19].

3.7 Критика

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

Глава 4

Прочее

Что хранится в объектных и бинарных (.o, .obj, .exe, .dll, .so) файлах?

Обычно только данные (глобальные переменные) и тела ф-ций (включая методы классов).

Информации о типах (классы, структуры, typedef(1.2.1)-ы) там нет. Это может помочь в понимании того как всё устроено.

Читайте также о name mangling: (3).

В этом заключается одна из больших проблем декомпиляции — отсутствие информации о типах.

О том как всё компилируется в машинный код, подробнее можно почитать здесь: [19].

4.1 Возврат кодов ошибок

Самый простой способ сообщить вызываемой ф-ции о своем успехе, это вернуть булево значение, *false* — в случае ошибки, и *true* в случае успеха. Такого очень много в Windows API. А если нужно передать больше информации, то оставить код ошибки в TIB[1], откуда затем получить это значение вызвав GetLastError(). Либо, в UNIX-средах, оставлять код ошибки в глобальной переменной *errno*.

4.1.1 Отрицательные коды ошибок

Еще один интересный способ передать больше информации в возвращаемом значении. Например, в документации IBM DB2 9.1, мы можем увидеть такое:

> 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]

В исходниках Linux можно увидеть и такое [4]:

Листинг 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
```

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

```
 * @offset: offset in bytes into the page
 * @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;
}
```

Обратите внимание — ф-ция может вернуть как количество байт, так и код ошибки. Тип ssize_t это "знаковый" size_t, то есть, способный принимать отрицательные значения. ENOMEM и EFAULT это стандартные коды ошибок из errno.h.

4.2 Глобальные переменные

Мода на ООП[5] и всякие такие штуки утвердила что глобальные переменные это плохо, тем не менее, в каких-то разумных случаях, это можно использовать (не забывая о мультитредовости), например, для возврата большего количества информации из ф-ций.

Так, некоторые стандартные ф-ции Си возвращают код ошибки через глобальную переменную *errno*, которая, в наше время, фактически не глобальная, а находится внутри TLS.

В Windows API код ошибки можно узнать вызвав GetLastError(), которая, на самом деле, выдает значение из TIB.

В кодегенераторе компилятора OpenWatcom, всё находится в глобальных переменных, а самая главная ф-ция выглядит так:

Листинг 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;
```

[5]Объектно-Ориентированное Программирование

```
    #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();
                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 Битовые поля

Это очень популярная штука в Си, да и в программировании вообще.

Для задания булевых значений "true" или "false", можно передавать 1 или 0 в байте или в 32-битном регистре, или в типе *int*, но это очень не экономно в плане расхода памяти. Намного удобнее передавать такие значения в отдельных битах.

К примеру, стандартная ф-ция findfirst() возвращает структуру о найденном файле, где аттрибуты файла передаются такими флагами:

```
#define _A_NORMAL 0x00
#define _A_RDONLY 0x01
#define _A_HIDDEN 0x02
#define _A_SYSTEM 0x04
#define _A_SUBDIR 0x10
#define _A_ARCH 0x20
```

Конечно, передавать каждый аттрибут отдельной переменной типа *bool* было бы очень неэкономично.

И напротив, для указания флагов в ф-цию можно исползовать битовые поля. Например, CreateFile() [6] из Windows API.

Для задания флагов, чтобы не запутаться и не опечататься в значениях каждого бита, можно писать так:

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

(Компилятор все равно всё это легко соптимизирует).

А чтобы было удобнее проверять/выставлять/удалять отдельный бит/флаг, можно использовать подобные макросы:

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

С другой стороны, необходимо помнить, что операции выделения отдельного бита из значения типа *int* обычно даются CPU "дороже", чем работа с типом *bool* в 32-битном регистре. Так что, если скорость для вас намного критичнее чем экономия памяти, можно попробовать использовать тип *bool*.

4.4 Интересные open-source проекты для изучения

4.4.1 Си

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

4.4.2 Си++

- 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

Глава 5

Инструменты от GNU

5.1 gcov

(*Coverage*) Позволяет показать исполненные строки кода и количество:

Попробуем простой пример:

```
#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);
};
```

Компилируем так (-g означает добавление отладочной информации к выходному исполняемому файлу, -O0 — отсутствие оптимизации кода [1], остальное — параметры для gcov):

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

GCC вставляет в код ф-ции собирающие статистику. Конечно, это замедляет работу самой программы. После исполнения, появляются файлы gmon.out, gcov_test.gcda, gcov_test.gcno.

Запускаем gcov:

```
gcov gcov_test
```

Получим в результате текстовый файл gcov_test.c.gcov:

Листинг 5.1: gcov_test.c.gcov

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

[1] Это важно потому что генерируемые инструкции для процессора должны быть сгруппированны и соответствовать строкам на Си/Си++. Оптимизация может сильно исказить эту связь и gcov (и еще и gdb) не сможет корректно показывать строки исходника.

```
        -:    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:
```

Строки маркированные ##### не исполнялись. Это может быть очень полезным для составления тестов.

Глава 6

Тестирование

Тестирование это крайне важно. Самый простой тест это некая программа, вызывающая ваши ф-ции и сверяющая результаты с корректными:

```
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);
};
```

Тесты должны работать автоматически (без внешнего вмешательства) и запускаться как можно чаще, очень хорошо если после каждого изменения кода.

Для составления тестов очень полезен gcov (5) либо иной инструмент для вывода покрытия (coverage). Хороший тест должен проверять работоспособность всех ф-ций, а также всех частей ф-ций.

Другие статьи и советы о тестировании: [13].

Послесловие

6.1 Вопросы?

Совершенно по любым вопросам, вы можете не раздумывая писать автору: <dennis@yurichev.com>

Пожалуйста, присылайте мне информацию о замеченных ошибках (включая грамматические), и т.д.

Список принятых сокращений

STL Standard Template Library . . . 39

TLS Thread Local Storage . . . 22

TIB Thread Information Block . . . 53

RAII Resource Acquisition Is Initialization . . . 26

ОС Операционная Система . . . i

ЯП Язык Программирования . . . 12

ООП Объектно-Ориентированное Программирование . . . 54

ГПСЧ Генератор псевдослучайных чисел . . . 48

MSVC Microsoft Visual C++ . . . 3

GCC GNU Compiler Collection . . . 3

POSIX Portable Operating System Interface . . . 39

CPU Central Processor Unit . . . 3

IDE Integrated development environment . . . 6

RISC Reduced instruction set computing . . . 3

IOCCC The International Obfuscated C Code Contest . . . 12

GUID Globally Unique Identifier . . . 42

UUID Universally unique identifier . . . 42

CRT C runtime library . . . 23

CSV Comma-separated values . . . 35

CPU Central processing unit . . . 3

Литература

[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. Введение в reverse engineering для начинающих. 2013. Also available as http://yurichev.com/writings/RE_for_beginners-ru.pdf.

Словарь терминов

Интегральный тип Тип данных приводимый к челочисленному числу, например: int, short, char. 4, 10, 21, 22

Итератор Указатель на текующий элемент списка или иной коллекции, используется для перебора этих элементов. 1, 7, 8, 13, 50

Свободная функция Ф-ции не являющиеся методом какого-либо класса. 51

BigInt Так обычно называют библиотеки для работы с числами произвольной точности, например `http://gmplib.org/`. 42, 52

glibc Стандартная библиотека в Linux. 18, 44

Предметный указатель