

# Конспекты по C++

## Семестр 2

Марк Тюков, Кудрявцев Иван

Весна 2020 года

## Содержание

<b>1</b>	<b>Введение в язык</b>	<b>3</b>
1.1	Инструкции . . . . .	3
1.1.1	Declarations (объявления) . . . . .	3
1.1.2	Expressions . . . . .	3
1.1.3	Control statements . . . . .	4
1.2	Ошибки компиляции СЕ . . . . .	4
1.2.1	Лексические ошибки . . . . .	4
1.2.2	Синтаксические ошибки . . . . .	4
1.2.3	Семантические ошибки . . . . .	4
1.3	Ошибки выполнения RE - Runtime Error . . . . .	4
1.3.1	Segmentation fault - обращаемся к чужой памяти . . . . .	4
1.4	Undefined Behaviour . . . . .	4
1.5	Linking error - ошибки линковщика . . . . .	5
<b>2</b>	<b>Указатели. Массивы</b>	<b>5</b>
2.1	Pointers, Arrays, functions, etc . . . . .	5
2.1.1	Pointers . . . . .	5
2.1.2	Операции с указателями . . . . .	5
2.2	Arrays . . . . .	6
2.2.1	Операции над массивами: . . . . .	6

2.3	Functions . . . . .	6
2.3.1	Overloading resolution . . . . .	6
2.3.2	Указатель на функцию . . . . .	7
2.3.3	Аргументы по умолчанию . . . . .	7
2.3.4	Inline . . . . .	7
<b>3</b>	<b>Статическая и динамическая память</b>	<b>7</b>
3.1	Статическая память . . . . .	7
3.1.1	static variables . . . . .	7
3.2	Динамическая память - выдается по требованию .	7
3.2.1	операторы . . . . .	7
3.2.2	Переменная . . . . .	8
3.2.3	Массив . . . . .	8
3.2.4	Некорректное использование delete . . . .	8
<b>4</b>	<b>Ссылки</b>	<b>8</b>
4.1	Создание ссылки . . . . .	8

# 1 Введение в язык

## 1.1 Инструкции

### 1.1.1 Declarations (объявления)

#### Переменные

```
1 type id [= value];
```

**Примеры** ключается в стек [unsigned] int/long long/char/float/double/double/bool

P.S.: size\_t  $\equiv$  unsigned long long

#### Функции, структуры

```
1 void f(int x, double y){}  
2 struct S{};
```

P.S.: using vi = std::vector<int>;

**Declaration vs definition** !!! One definition rule (ODR)

### 1.1.2 Expressions

#### Базовые операторы

1. Арифметические операторы (+, -, \*, /, %)
2. Побитовые операторы (&, |, ^, ~, <<, >>)
3. Логические операторы (&&, ||, !)
4. Операторы сравнения (==, <, >, <=, >=)
5. Assignments (=, + =, - =, \* =, / =, % =, & =, | =, ^ =, << =, >> =)

6. Инкремент и декремент ( $++x$ ,  $x++$ )
7. `sizeof()` - возвращает число, которое нужно для хранения входных данных (в байтах). Он не сохраняет результат операций. Например, `sizeof(x++)` не изменит  $x$
8. Тернарный оператор " $\dots ? \dots : \dots$ "
9. Запятая – выполняет левую часть, потом правую, возвращает правую. Она гарантирует, что левая часть закончит выполнение до того, как начнет выполняться правая.

### 1.1.3 Control statements

1. `if (statement) else`
2. `while(statement)`
3. `for (declaration or expression; bool expression; expression)`

## 1.2 Ошибки компиляции СЕ

### 1.2.1 Лексические ошибки

неизвестный символ

### 1.2.2 Синтаксические ошибки

`if = 5)`

### 1.2.3 Семантические ошибки

...

## 1.3 Ошибки выполнения RE - Runtime Error

### 1.3.1 Segmentation fault - обращаемся к чужой памяти

1. заканчивается стек рекурсии

## 1.4 Undefined Behaviour

Когда пишем что-то такое, на что компилятор в стандарте не имеет четкой инструкции.

1. обращение к массиву по несуществующему индексу.

```
1 int a[10];  
2 a[100];
```

В таком случае можно получить как RE, так и UB

**Если в программе случился UB, то не гарантируется ничего!**

$(1 \neq 0)$  будет true

## 1.5 Linking error - ошибки линковщика

После компиляции, например что-то не объявлено.

**Насколько плохи все эти ошибки?**

**СЕ** - компилятор наш друг

**РЕ** - **плохо**, сервер может упасть / может случиться что-то плохое во ВРЕМЯ ВЫПОЛНЕНИЯ

**UB** - **ужасно!** не совершайте преступление, не делайте UB!

*UB и RE компилятор не блокирует зачастую при компиляции.*

## 2 Указатели. Массивы

### 2.1 Pointers, Arrays, functions, etc

#### 2.1.1 Pointers

```

1  int x
2  {
3      int y; // выделение памяти
4  } // — удаление из памяти на( самом деле потеря адреса. Что
      происходит с данными по тому адресу — хз)
5
6  type* p;
7  *p; // — унарная звездочка разыменовывания.
8
9  type* p = &x; // — кладем в p адрес x
10 p+1; p-1;

```

## 2.1.2 Операции с указателями

1. Разыменовывание
2. Сложение с числами
3. Разность указателей

**void\*** - указатель на несуществующую область памяти. Такие указатели нельзя увеличивать и вычитать друг из друга

Но! указатели **можно преобразовывать**

**nullptr** - константный указатель (типа *nullptr\_t*) - аналог нуля для указателей

**\*nullptr** // - UB :)

## 2.2 Arrays

type a[10] - выделение памяти на стэке для 10 элементов

### 2.2.1 Операции над массивами:

1.  $*(a+i)$  Возьми адрес, где начинается массив, прибавь к нему число  $i$  и разыменуй.
2. Array to pointer conversion  
`type* b = a;`
3. `sizeof(a) = размер массива * sizeof(тип)`  
`sizeof(type*) != sizeof(a);`

## 2.3 Functions

**Сигнатура** - набор типов принимаемых аргументов.

Можно объявить несколько функций с разными сигнатурами

```
1 type f(int);  
2 type f(double);  
3 type f(int, int);
```

Эти функции могут возвращать данные разных типов

Компилятор при вызове таких функция совершает разрешение перегрузки (**overloading resolution**) - принятие решение о выборе версии функции

### 2.3.1 Overloading resolution

1. В точности такой тип
2. Преобразование в `int`
3. Если не получается однозначно выбрать, будет ошибка компиляции **Ambiguous call**

Пример `f(float)` вызываем, когда есть только от `double` и от `int`

**Читать в стандарте!!!**

### 2.3.2 Указатель на функцию

```
1 int f(double);  
2 int (*)(double) pf = f;  
3  
4 type f() {  
5 }
```

**P.S.** Запятая при указании аргументов - устойчивая конструкция для аргументов, а не *Expressions*

```
int a = 5;
```

Здесь знак равно - это **не оператор присваивания**, а устойчивая конструкция инициализации!

### 2.3.3 Аргументы по умолчанию

Аргументы по умолчанию функций указываются **в конце**

```
f(double x, int n = 10)
```

### 2.3.4 Inline

Непосредственная **вставка кода** в указанное место **при компиляции**

**inline** - лишь **рекомендации** компилятору (компилятор решает сам, он умный)

## 3 Статическая и динамическая память

### 3.1 Статическая память

#### 3.1.1 static variables

**Свойства**

1. один раз заводятся
2. размер вычисляет компилятор до запуска программы



3. инициализируются один раз
4. значение при разных вызовах функции сохраняются

## 3.2 Динамическая память - выдается по требованию

### 3.2.1 операторы

`new, delete`

### 3.2.2 Переменная

`type* p = new type();` - запрашиваем память  
потом нужно **освободить**  
`delete p;`

### 3.2.3 Массив

`type* p = new type[n];` - запрашиваем память  
`delete[] p;`

`delete` и `new` - expressions

### 3.2.4 Некорректное использование `delete`

1. `Delete` не на тот указатель - UB
2. `delete` без `[]` для массивов - UB
3. Если не освободить память возможны **memory leak**
4. Если дважды удалить, то будет RE(Seg.F.)

**P.S.**

`delete p, pp;`

Это **expression**. Парсится по запятой. Выполняется `delete p`.  
Потом `pp.pp` Не удалится ;

## 4 Ссылки

### 4.1 Создание ссылки

type x;

type& y = x; - Новое название переменной. y - **ссылка** на x.

```
1 swap(int x /* — создаем локальную КОПИЮ икса */, int y){
2     int t = x;
3     x = y;
4     y = t;
5 } // — так не работает.
6
7 type x;
8 type y = x; // — Создаем ссылку. НО НЕ В C++. В Java — да
9
10 type x;
11 type& y = x; — Новое название переменной. Y — ссылка на x.
```

Отныне **нет способа отличить** y от x. Отныне и **навсегда**  
*"Я поступил на физтех- "Я поступил в МФТИ"*  
fewnewkrliе