

# Конспекты по C++.

## Семестр 2

Марк Тюков, Иван Кудрявцев

Весна 2020 года

## Содержание

<b>1</b>	<b>Введение в язык</b>	<b>5</b>
1.1	Инструкции . . . . .	5
1.1.1	Declarations (объявления) . . . . .	5
1.1.2	Expressions . . . . .	5
1.1.3	Control statements . . . . .	5
<b>2</b>	<b>Ошибки</b>	<b>6</b>
2.1	Compilation Error . . . . .	6
2.1.1	Лексические ошибки . . . . .	6
2.1.2	Синтаксические ошибки . . . . .	6
2.1.3	Семантические ошибки . . . . .	6
2.2	Runtime Error . . . . .	6
2.2.1	Segmentation Fault . . . . .	6
2.3	Undefined Behaviour . . . . .	6
2.4	Linking Error . . . . .	6
2.5	Насколько плохи все эти ошибки? . . . . .	6
<b>3</b>	<b>Указатели. Массивы</b>	<b>7</b>
3.1	Pointers, arrays, functions, etc . . . . .	7
3.1.1	Pointers . . . . .	7
3.1.2	Операции с указателями . . . . .	7
3.2	Arrays . . . . .	7
3.2.1	Операции над массивами: . . . . .	7
3.3	Functions . . . . .	8
3.3.1	Overloading resolution . . . . .	8
3.3.2	Указатель на функцию . . . . .	8
3.3.3	Аргументы по умолчанию . . . . .	8
3.3.4	Inline . . . . .	8
<b>4</b>	<b>Память</b>	<b>9</b>
4.1	Статическая память . . . . .	9
4.1.1	Static Variables . . . . .	9
4.2	Динамическая память . . . . .	9
4.2.1	Операторы . . . . .	9
4.2.2	Переменная . . . . .	9

4.2.3	Массив . . . . .	9
4.2.4	Некорректное использование delete . . . . .	9
<b>5</b>	<b>Ссылки</b>	<b>10</b>
5.1	Создание ссылки . . . . .	10
5.2	Операции над ссылками: . . . . .	10
<b>6</b>	<b>Константы</b>	<b>11</b>
6.1	Объявление . . . . .	11
6.2	Что можно и нельзя . . . . .	11
6.3	Константный указатель . . . . .	11
6.4	Константная ссылка . . . . .	11
<b>7</b>	<b>Приведение типов</b>	<b>12</b>
7.1	static_cast . . . . .	12
7.2	Три запрещенных заклинания: . . . . .	12
<b>8</b>	<b>Введение в ООП</b>	<b>13</b>
8.1	Что это такое? . . . . .	13
8.2	Три волшебных слова в ООП . . . . .	13
<b>9</b>	<b>Инкапсуляция</b>	<b>13</b>
9.1	Классы и структуры . . . . .	13
9.2	Модификаторы доступа . . . . .	13
9.3	Оператор “стрелочка” . . . . .	13
9.4	Указатель this . . . . .	14
9.5	Конструкторы . . . . .	14
9.6	Деструкторы . . . . .	15
9.7	Копирование, копирующее присваивание и правило трех . . . . .	15
9.8	Константные методы . . . . .	16
9.9	Списки инициализации в конструкторах . . . . .	17
9.10	Friends . . . . .	17
9.11	Explicit . . . . .	17
9.12	Contextual conversial . . . . .	17
9.13	Статические поля и методы . . . . .	18
9.14	Pointers to members . . . . .	18
<b>10</b>	<b>Inheritance (наследование)</b>	<b>18</b>
10.1	Модификатор PROTECTED . . . . .	18
10.2	Размещение в памяти объектов наследников . . . . .	19
10.3	Поиск имён при наследовании . . . . .	19
10.4	Multiple inheritance . . . . .	20
10.5	Virtual наследование . . . . .	21
10.5.1	Private virtual . . . . .	21
10.5.2	Очередное запрещённое заклинание . . . . .	21
10.6	Приведение типов при наследовании . . . . .	21
10.6.1	static_cast . . . . .	22
10.6.2	reinterpret_cast . . . . .	22
10.7	Виртуальные функции . . . . .	23
10.8	Override . . . . .	23

10.8.1	Полиморфизм . . . . .	24
10.9	<code>dynamic_cast</code> . . . . .	24
10.10	Virtual destructor . . . . .	24
10.11	Pure virtual functions and abstract classes . . . . .	24
<b>11</b>	<b>Templates (шаблоны)</b>	<b>25</b>
11.1	Синтаксис . . . . .	25
11.2	Typedef . . . . .	25
11.3	Перегрузка шаблонной функции . . . . .	25
11.4	Специализация шаблонов . . . . .	26
11.4.1	Частичная специализация . . . . .	26
11.4.2	Порядок специализаций . . . . .	26
11.4.3	template template parameters – шаблонный параметры . . . . .	27
11.5	Метафункции . . . . .	27
11.6	Неоднозначность . . . . .	28
11.7	Шаблоны с переменным количеством элементов . . . . .	28
11.7.1	Синтаксис самого общего случая . . . . .	28
11.7.2	Пример . . . . .	28
11.7.3	Переменное количество аргументов заданного типа . . . . .	28
11.7.4	Количество переданных аргументов . . . . .	29
<b>12</b>	<b>Последовательные контейнеры</b>	<b>29</b>
12.1	Vector . . . . .	29
12.1.1	Объявление и прочие операции . . . . .	29
12.1.2	Реализация . . . . .	30
12.1.3	Вектор <code>bool</code> . . . . .	30
12.2	Deque . . . . .	31
12.3	List . . . . .	31
<b>13</b>	<b>Ассоциативные контейнеры</b>	<b>31</b>
13.1	Map . . . . .	31
13.1.1	Некоторые операции . . . . .	31
13.2	Unordered map . . . . .	32
<b>14</b>	<b>Exceptions (Исключения)</b>	<b>32</b>
14.1	Общий синтаксис . . . . .	32
14.2	Как кидать исключение . . . . .	32
14.3	Дополнительный синтаксис . . . . .	32
14.4	Метод <code>what()</code> . . . . .	33
14.5	Memory leaks caused by exceptions . . . . .	33
14.5.1	Исключения в конструкторах . . . . .	34
14.5.2	Исключения в деструкторах . . . . .	34
14.6	Спецификация исключений . . . . .	35
14.6.1	Оператор <code>noexcept</code> . . . . .	35
14.7	function-try block . . . . .	36
14.8	Finally in Try-Catch . . . . .	36

<b>15 Iterators (Итераторы)</b>	<b>36</b>
15.1 Categories of iterators . . . . .	36
15.1.1 Input . . . . .	36
15.1.2 Forward . . . . .	36
15.1.3 Bidirectional . . . . .	36
15.1.4 Random access . . . . .	36
15.1.5 Contiguous . . . . .	36
15.2 Iterator Traits . . . . .	37
15.3 Distance and Advance . . . . .	37
15.4 Standart algorithms . . . . .	37
15.5 Output Iterators . . . . .	37
15.6 Insert, Back_insert and Front_insert iterator . . . . .	38
15.7 Инвалидация итераторов . . . . .	38
15.8 Auto, decltype . . . . .	38
<b>16 Move semantics и RVALUE ссылки</b>	<b>39</b>
16.1 Что это такое и мотивация . . . . .	39
16.2 Magic function <code>std::move()</code> . . . . .	39
16.3 RValue ссылки . . . . .	39

# 1 Введение в язык

## 1.1 Инструкции

### 1.1.1 Declarations (объявления)

#### Переменные

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

#### Примеры

[unsigned] int/long long/char/float/double/long 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 "... ? ... : ..."
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)`

## 2 Ошибки

### 2.1 Ошибки компиляции (Compilation Error)

#### 2.1.1 Лексические ошибки

Неизвестный символ

#### 2.1.2 Синтаксические ошибки

if = 5)

#### 2.1.3 Семантические ошибки

...

Семантическая ошибка (или ещё «смысловая») возникает, когда код синтаксически правильный, но выполняет не то, что нужно программисту.

### 2.2 Ошибки выполнения RE - Runtime Error

#### 2.2.1 Segmentation Fault - обращаемся к чужой памяти

1. заканчивается стек рекурсии

### 2.3 Undefined Behaviour

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

**Пример:** 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 != 0) будет true

### 2.4 Linking error - ошибки линковщика

После компиляции, например, что-то не объявлено.

### 2.5 Насколько плохи все эти ошибки?

CE - компилятор наш друг

RE - 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.

## 3 Указатели. Массивы

### 3.1 Pointers, arrays, functions, etc

#### 3.1.1 Pointers

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

#### 3.1.2 Операции с указателями

1. Разыменование;
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

P.S.:

```
1 *nullptr // UB :)
```

## 3.2 Arrays

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

#### 3.2.1 Операции над массивами:

1.  $*(a+i)$  — взятие адреса, где начинается массив, прибавление к нему числа  $i$  и разыменование.
2. Array-To-Pointer conversion

```
type* b = a;
```

3. `sizeof(a)` — размер массива \* `sizeof(тип)`  
`sizeof(type*)`  $\neq$  `sizeof(a)`;

### 3.3 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 функций совершает разрешение перегрузки (**overloading resolution**), то есть принимает решение о выборе версии функции

#### 3.3.1 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 `double` и `int`.

**P.S.:** Читать в стандарте!!!

#### 3.3.2 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 **expression**.  
`int a = 5;`

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

#### 3.3.3 Аргументы по умолчанию

Аргументы по умолчанию функций указываются в конце.

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

#### 3.3.4 Inline

Непосредственная вставка кода в указанное место при компиляции.

`inline` — лишь рекомендации компилятору (компилятор решает сам, он умный)



## 4 Статическая и динамическая память

### 4.1 Статическая память

#### 4.1.1 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.

### 4.2 Динамическая память - выдается по требованию

#### 4.2.1 Операторы

`new`, `delete`

#### 4.2.2 Переменная

```
1 type* p = new type(); // запрашиваем память
```

Потом нужно освободить память

```
1 delete p;
```

#### 4.2.3 Массив

```
1 type* p = new type[n]; // запрашиваем память
2 delete[] p;
```

**P.S.:** `delete` и `new` - expressions

#### 4.2.4 Некорректное использование `delete`

1. `delete` не на тот указатель - UB
2. `delete` без `[]` для массивов - UB
3. Если не освобождать память возможен **Memory Leak**
4. Если дважды удалить, то будет RE (Segmentation Fault)

**P.S.:** `delete p, pp;` — это **expression**. Парсится по запятой. Выполнится `delete p`, потом `pp` (просто обращение). То есть `pp` не удалится :(

## 5 Ссылки

### 5.1 Создание ссылки

type x;

type& y = x; // новое название переменной, y - **ссылка** на x.

Не заработает:

```
1 void swap(int x /* создаем локальную КОПИЮ икса */, int y){
2     int t = x;
3     x = y;
4     y = t;
5 }
```

```
1 type x;
2 type y = x; // Создаем ссылку, но НЕ в C++. В Java - да
```

**P.S.:** Java  $\equiv$  ♥

```
1 type x;
2 type& y = x; // новое название переменной. y - ссылка на x. Всё, что делается с y делается и с x
```

Отныне нет способа отличить y от x. Отныне и навсегда:

*“Я поступил на физтех”  $\equiv$  “Я поступил в МФТИ”*

Правильная реализация swap:

```
1 void swap(int& x, int& y) {
2     int t = x;
3     x = y;
4     y = t;
5 }
```

### 5.2 Операции над ссылками:

(В основном проблемы :))

1. Можно всё то, что можно делать с x.
2. Нельзя не инициализировать.
3. Нельзя делать ссылки на **rvalue**:

```
1 int& x = 5;
```

4. Можно:

```
1  int x;  
2  int& f() {  
3      return x; // x – глобальный  
4  }
```

5. Если x в предыдущем примере локальный, то так нельзя, будет битая ссылка (**Dangling Reference**)

## 6 Константы

### 6.1 Объявление

```
const int x = 3;
```

Это такой тип, к которому применены только константные операции

### 6.2 Что можно и нельзя

1. Необходимо инициализировать сразу при объявлении!!!
2. Можно передавать не константную версию туда, где нужна константная

### 6.3 Константный указатель

```
const int* p = new int // указатель на константный инт: *p = 1; – нельзя; можно p++  
int* const p = new int // искомый константный указатель
```

### 6.4 Константная ссылка

Нельзя `int& const x = 1;` // это какая-то фигня. Не надо так :(

Можно заводить ссылки на константные переменные, но не стоит делать константные ссылки (будет либо ошибка, либо не будет иметь смысла)

```
1  int x = 1;  
2  const int& y = x;
```

Выше мы можем менять x, но не через y.

**Продление жизни ссылок** `const int& x = 5` // можно инициализировать rvalue  
Такая ссылка не умрет, пока не закончится локальная видимость переменной.<sup>1</sup>

---

<sup>1</sup><https://habr.com/ru/post/186790/>

## 7 Приведение типов

### 7.1 static\_cast

```
1 static_cast<type> (expression);
```

Если преобразование не однозначно, то будет СЕ (неоднозначный каст).

Будет СЕ, если нет подходящего преобразования.

Название static из-за того, что всё делается на уровне компиляции.

Не знаешь, какое приведение типов тебе нужно — тебе нужен static\_cast

### 7.2 Три запрещенных заклинания:

**Первое заклинание:** reinterpret\_cast<>(...)

Берёт объект как байты в памяти и начинает считать, что это другой тип.

Можно reinterpret\_cast указателей:

```
reinterpret_cast<type*>(...);
```

reinterpret\_cast ссылки:

```
type y = reinterpret_cast<type*>(x);
```

**Второе заклинание:** const\_cast<>(...)

Означает “перестань считать константу константой” (которое неявно запрещено) и наоборот. Вообще, это UB.

*Пример:*

```
1 int& z = const_cast<int*>(y);
```

Теоретически, это нужно, когда есть две функции:

```
1 f(int f)
2 f(const int&)
```

Если по какой-то причине захотим запустить f() от int’а по пути константы (если f для них работает совершенно по разному), то нужно будет привести int к const int.

const\_cast — это угнетение компилятора.

**Третье заклинание** C-style cast

Пытается сделать всё, чтобы получилось. Так что мы даже не узнаем, что сделалось

```
1 (type)(expression)
```

## 8 Введение в ООП

### 8.1 Что это такое?

Код состоит из объявления разных объектов некоторых типов и expressions с этими объектами.

### 8.2 Три волшебных слова в ООП

Инкапсуляция, наследование, полиморфизм — основные принципы, на которых основано ООП. Далее мы подробнее изучим каждое слово, а также увидим связь одного с другим. В частности, по мере изучения нового слова могут быть сделаны уточнения и нововведения в предыдущую тему.

## 9 Инкапсуляция

Точное определение инкапсуляции дать сложно. Ниже приведены два определения на не совсем формальном языке, дабы понять суть данного понятия.

**Инкапсуляция** - оборачивание в (защитную) оболочку внутренней реализации с помощью ограничивающего интерфейса.

**Инкапсуляция** - совместное хранение полей и методов (но ограниченный доступ к ним извне).

### 9.1 Объявление классов и структур

(авторы ленивые, поэтому описание некоторого базового синтаксиса может быть пропущено)

```
1 class C {
2     /* тело класса */
3 };
4
5 struct S {
6     /* тело структуры */
7 };
```

В теле — методы, поля и т.д.

### 9.2 Модификаторы доступа

```
1 class C {
2     private: // может быть опущено
3     public:  // дальше всё public, пока не встретится модификатор доступа
4     ...
5     protected:
6     ...
7 };
```

Для структуры всё то же самое, только изначально всё публично, а не приватное.

### 9.3 Оператор “стрелочка”

$(*p).f(); \equiv p \rightarrow f();$

## 9.4 Указатель this

...

## 9.5 Конструкторы

Конструктор нужен для инициализации объектов некоторого класса.

```

1  C x = ... ;
2  // варианты инициализации
3  C x (...);
4  C x {...};
5  C x = C (...);
6  C x = {...}; // так еще можно делать в структуре без конструктора (если все поля публик) – агрегатная
  инициализация.

```

Дальше в какой-то степени будем реализовывать класс строк

```

1  class String {
2  public:
3
4      String() { // конструктор
5          str = new char[16]; // начальный размер
6          size = 0;
7      }
8
9      String(size_t n) {
10         ...
11     }
12
13 private:
14     char* str;
15     size_t size;
16 };

```

Конструктор по умолчанию – такое компилятор может сделать сам (он это делает, если мы этого не сделали), но он будет инициализировать по умолчанию все поля (и это в большинстве случаев плохо).

Первое правило – компилятор сам создает конструктор, если мы его не определили и не определили никакой другой конструктор. Но можно явно попросить его сгенерировать такой.

```

1  String() = default; // начиная с C++11

```

Конструкторы нужны в тех случаях, когда надо инициализировать не тривиально.

Можно поле создавать так (начиная с **C++11**):

```

1  size_t size = 0; // это будет дефолтная инициализация.

```

Можно сделать делегацию конструктора. Сначала выполнится один конструктор, потом другой.

```

1  String(...) : String(...) {
2      ... // дополнительный код
3  }

```

Если структура состоит только из

## 9.6 Деструкторы

При запуске деструктора:

1. Удаляем нетривиальные объекты (у которых выделена память оператором *new*);
2. Закрытие потоков;
3. Освобождение ресурсов;

**P.S.:** Всегда надо подчищать за собой!

Деструктор **нельзя** сделать приватный.

Деструктор можно вызывать явно (но в крайних и очень редких случаях).

```
1 ~String() {
2     delete [] str;
3 }
```

Все, что нужно делать в деструкторе – нетривиальные действия. Все остальные поля уничтожаются сами после выхода из деструктора.

## 9.7 Копирование, копирующее присваивание и правило трех

Для большинства объектов хочется уметь делать копии.

```
1 S s;
2 S s1 = s; // если конструктор копирования нет, компилятор его автоматически создает, просто копируя все
           поля.
```

Это плохо с нетривиальными полями (ссылки, указатели и т.д.) – может быть RE.

**Инициализация конструктора копирования** Важно делать `const String& s`

```
1 String(const String& s) {
2     str = new char[s.size];
3     for (...) {
4         ...
5     }
6 };
```

Аналогично обычным конструкторам можно написать `= default`

Чтобы запретить копирование:

1. Сделать приватным
2. `String(const String& s) = delete;` начиная с C++11

Если хотим заменить уже существующий объект на другой, то нужно удалить старый объект и положить туда новый.

Тривиальный оператор присваивания генерируется автоматически.

Он должен возвращать результат присваивания (того же типа). Хотим возвращать *lvalue*. Возвращаем неконстантную ссылку, чтобы можно было ей присваивать

Если хотим написать

```
1 String s;
2 String s1;
3 s1 = s;
```

то здесь будет вызываться оператор присваивания.

```
1 String& operator =(const String& s) {
2     // стоит проверять на присваивание самому себе
3     if (this == &s) return *this;
4     delete [] str;
5     ...
6     return *this;
7 }
```

Оператор присваивания тоже можно писать через `= default`

**Rule of three** Если в нашем классе потребовалось реализовать что-то одно из нетривиальных конструктора, деструктора или оператора присваивания, то потребуются и все три.

## 9.8 Константные методы

Такие методы, что их можно выполнять над константными переменными. Надо писать слово `const`, когда метод ничего не меняет.

```
1 void f(...) const {
2     ...
3 }
```

Если мы хотим завести счетчик, сколько раз метод был вызван, а метод константный. В таком случае, если слово “anticonst” – `mutable`

То есть счетчик будет реализован:

```
1 mutable int counter;
```

```
1 char& operator [] (size_t n);
2 const char& operator [] (size_t n) const;
```



## 9.9 Списки инициализации в конструкторах

```
1 struct S {  
2     int& x;  
3     const int y;  
4  
5     S(int& x, int y) { // когда вошли в эту область видимости, поля должны быть уже проинициализированы,  
6         // а ссылку невозможно проинициализировать так  
7     }  
8     // Вместо этого так:  
9  
10    S(int& x, int y) : x(x), y(y) {} // здесь инициализация будет до входа в конструктор  
11 };
```

Списки инициализации сохраняют нам одно копирование.

## 9.10 Friends

Иногда захочется, чтобы приватное поле было доступно.

```
1 friend void f(int);  
2 friend class C;
```

```
1 friend istream& operator >> (istream& in, S& x)
```

## 9.11 Explicit

Если у нас большой код, то велика возможность что-то пропустить и получить неявное преобразование там, где его не должно быть. Для этого можно запретить неявную конвертацию.

```
1 explicit String(size_t n); // можно вызывать только явно
```

Операторы преобразования тоже могут быть *explicit* (с C++11)

```
1 explicit operator int() { // оператор преобразования к инту  
2     ...  
3     return x; // x типа int  
4 }
```

## 9.12 Contextual conversial

Это конверсия в буль в ифах, форах, вайлах (под условиями). Такая конверсия игнорирует `explicit` (потому что не является неявным преобразованием)

### 9.13 Статические поля и методы

Это те поля и методы, которые относятся не к конкретному объекту, а ко всему классу в целом.

- Память на них выделяется при компиляции (в статической области).
- Из статических методов есть доступ только к статическим полям.
- Если статический метод публичный, то для вызова его извне класса надо писать:

```

1  class C {
2  public:
3      static void method();
4  }
5
6  int main() {
7      C object;
8      object.method() // неправильно
9
10     C::method() // правильно
11 }
```

### 9.14 Pointers to members – указатель на член класса

```

1  int S::* p = &S::*x; // для поля
2  int (*параметры*) (S::*) ... // для метода
3
4  S s;
5  s.*p // вернет ссылку на x // здесь .* - отдельный оператор
```

Пример: есть ориентированный граф и мы можем делать обход либо по обычному, либо по инвертированным ребрам. Хотим написать (одну) функцию, которая будет делать обход (как обычный, так и инвертированный). В зависимости от того, какой обход требуется, нужно завести указатель на начало и указатель на конец, а в обходе вызываем от указателя.

## 10 Inheritance – второй принцип ООП

Некоторые типы могут быть “подтипами” других. Производные типы содержат все поля и методы родителей, а также и некоторые свои.

**Синтаксис:**

```

1  class Derived : public /*private, protected*/ Base {
2      ...
3  }
```

### 10.1 Модификатор PROTECTED

Будет доступен членам, друзьям, детям (наследникам)

Стоит обращать внимание на тип, который используется (структура или класс)

## 10.2 Размещение в памяти объектов наследников

```

1 struct Base {
2     int a;
3
4     Base(int a){}
5
6 };
7
8 struct Derived : public Base{
9     int a;
10    int b;
11 };

```

sizeof(Derived) даст 3: a, a, b

```

1 Derived d;
2 d.a; // поле Derived
3 d.Base::a; // поле Base

```

То есть при создании наследника всегда создается родитель (со всеми полями и т.п.), а также сам класс, со всеми его полями. Также при удалении: сначала сам класс, потом родитель.

```

1 Derived(int a, int b, int c) : Base(a), a(b), b(c) {
2     ...
3 }

```

Циклическая объявление – ошибка компиляции.

**P.S.:** Когда пишем деструктор – не нужно удалять Base !!!

## 10.3 Поиск имён при наследовании

```

1 struct Granny {
2     int x;
3     void f();
4 }
5 struct Mom : private Granny {
6     int d;
7     void f(int y);
8 }
9 struct Son : public Mom {
10    int e;
11    void f(double y)
12 }
13
14 Son s;
15 s.f(1); // Тут произойдёт неявный каст в double

```

Другие сигнатуры функций будут не видны (*invisible*). Другие будут затменены сигнатурой из Son.

visible  $\neq$  accessible

Видимые – те, которые находит поиск имен. Доступные – те, к которым есть доступ по модификаторам доступа при наследовании.

А если сделать `private void f(double y)` внутри `Son`, то будет СЕ

Решение: **Qualife id**

```
1 s.Mom::f(1);
```

**P.S.:** Поиск имён происходит всегда до проверки доступа!

```
1 s.f(); // Ошибка компиляции, т.к. такая функция invisible или т.к. она private
2 s.Mom::f() // то же самое
3 s.Granny::f() // СЕ, т.к. имя Granny inaccessible
```

```
1 ...
2
3 class Son : public Mom {
4     public void f(double) {
5         Granny g; // не работает
6     }
7     public void f(double) {
8         ::Granny g; // работает
9     }
10 }
```

Разрешим сыну общаться с бабушкой

```
1 ...
2
3 class Mom : private Granny{
4     friend class Son; // разрешаем сыну общаться с бабушкой
5 }
6
7 ...
8 s.Granny::f() // по-прежнему нельзя
```

## 10.4 Multiple inheritance

ЭТО ПЛОХО. УЖАС. НЕ НАДО ТАК.

Почему? Из-за **проблемы ромбовидного наследования (diamond problem)**. Рассмотрим геометрические фигуры.

Имеем систему (стрелка ведет от сына к родителю)

Square  $\rightarrow$  Rectangle; Square  $\rightarrow$  Rombus; Rectangle  $\rightarrow$  Parallelogram; Rhombus  $\rightarrow$  Parallelogram;

Пусть в каждом лежит по инту. Тогда при создании квадрата создается два инта.

**Синтаксис:**

```
1 class Square : public Rhombus, public Rectangle {};
```

При множественном наследовании (если оно нам точно нужно), то нужно либо явно вызывать нужный метод (или обращаться к нужному полю), или следить, чтобы в предках не было методов с одинаковыми сигнатурами.

Ещё плохой пример:

Son→Mom; Mom→Granny; Son→Granny

Можно получить **inaccessible base class**, если захотим обратиться к чему-то из Granny. Из-за структуры у сына будет две бабушки, поэтому не понятно к какому имени мы обращаемся (то есть к полям бабушки).

## 10.5 Virtual наследование

```
1 class class_name1 : public virtual class_name {}
```

Когда хотим предотвратить создание объектов в множественном числе используется модификатор **virtual**

На самом деле при таком наследовании размер нового объекта будет даже больше! **int x** будет в единственном экземпляре, но будут указатели. Также нарушается порядок расположения в памяти: объект может лежать в памяти разрывным куском.

**Вставить рисунки**

На самом деле указатели будут указывать на некое специальное место, где компилятор создает хранилище информации для файлов такого типа

### 10.5.1 Private virtual

При приватном наследовании при попытке обращения к **x** будет ошибка уже из-за приватности

...

### 10.5.2 Очередное запрещённое заклинание

[[[G]M][G]S] Делаем

```
1 Granny granny = reinterpret_cast<Granny>(mom)
```

И тогда можем получить доступ к объекту **G** (самому левому)

## 10.6 Приведение типов при наследовании

Derived {int y, void f()}→Base {int y, void f()}

```

1  class Base {
2      int x;
3  }
4
5  class Derived : public Base {
6      int y;
7  }
8
9  Derived d;
10 Base* bp = &d;
11 Base& b = d;
12 b.f() // из Base
13 d.x++ // это изменит x и b, так как это разные имена одного и того же
14 Derived* dp = &b; // СЕ, потому что Derived потомок Base; чтобы так писать нужен явный каст
15 Derived& dd = b; // СЕ аналогично предыдущему
16
17
18 Base bb = d; // slicing

```

### 10.6.1 static\_cast

```

1  Derived& dd = static_cast<Derived&> b; // явный down cast (наследование вниз) (вверх делать тоже
    можно)

```

**P.S.:** Такое преобразование очень опасно, так как компилятор не может проверить данный процесс. Если в не лежал Derived, то получим UB.

Статик каст позволяет перестраховаться от нарушения наследования и нарушения приватности и очень помогает при множественном наследовании.

**Пример:**

Son→Mother; S→Father; Mother→Granny; Father→Granny

```

1  static_cast<Granny&>(s);

```

### 10.6.2 reinterpret\_cast

Данный каст будет игнорировать приватность и все прочее.

**Множественное наследование?**

```

1  Son s;
2  Father * fp = &s;
3  Father & f = s; //Происходит правильный сдвиг указателей
4  Father f = static_cast<Son> // Правильный сдвиг
5  Father f = reinterpret_cast<Son> // НЕправильный сдвиг

```

## 10.7 Виртуальные функции

Допустим, у нас есть координатная плоскость с двумя кругами: оба радиусом 1, в центре 1 и -1. Если мы опишем квадраты вокруг этих кругов, то равны ли эти квадраты?

Другой пример.  $\sin^2 x$  и  $f^2(x)$  Первое мы скорее поймем как произведение синусов, а второе — как композицию одинаковой функции. То есть для более узкого круга вещей (понятий) мы определяем одни и те же символы и формулы по разному.

Виртуальными функциями называются такие, у которых применяются более частные версии, даже если к объекту обратились как для общего вида.

```

1  class Base {
2      void f();
3      virtual void g();
4  };
5
6  class Derived : public Base {
7      void f();
8      virtual void g();
9  };
10
11  ...
12  Derived d;
13  Base& b = d;
14  b.f(); // вызывается из Base
15  b.g(); // вызывается из Derived

```

**P.S.:** виртуальные функции замедляют компилятор и время выполнения, потому что компилятору надо понимать что в данном случае мы хотим сделать. Он хранит указатель на таблицу виртуальных функций. За счет этого динамического подхода и теряется время.

Пусть

```

1  struct Base {
2      int a;
3      virtual void g();
4      void f();
5  }

```

`sizeof(Base) = 8` (без виртуальной функции — 4)

## 10.8 Override

```

1  class Base {
2      virtual void f() const;
3  }
4
5  class Derived: public Base {
6      void f();
7  }
8
9  Base& b = d;
10 b.f(); // вызовется из Base, так как мы забыли "const" в Derived

```

Функция в `Derived` не будет перегрузкой, а будет совершенно новой функцией.

Чтобы постоянно не копировать сигнатуру, то надо писать:

```
1  class Derived : public Base {
2      void f() const override;
3  }
```

Тогда если мы забудем что-то скопировать из сигнатуры, то будет СЕ и мы сможем подправить, чтобы программа нормально работала.

### 10.8.1 Полиморфизм

Это принцип, согласно которому можно использовать одно то же название функции, но получить разный результат в зависимости от типа объекта, над которым выполняется функция. Бывает статическим и динамическим.

Вообще, статическое – это то, что определяется на момент компиляции. Динамическое – в процессе работы программы.

Типы, у которых есть хоть одна виртуальная функция – **полиморфный тип** (*polymorphic type*).

## 10.9 dynamic\_cast

Рассмотрим пример с сыном, мамой, папой и бабушкой. У бабушки есть виртуальная функция `f()`.

```
1  Son s;
2  Mom& m = s;
3  // хотим сделать: Father& f = m; НО это СЕ!!! потому что тип мамы с типом папы не совместимы.
4  // тут на помощь приходит dynamic_cast
5  Father &f = dynamic_cast<Father&>(m) // залезь в таблицу виртуальных функций и сделай что надо ;)
```

**P.S.:** Если у `dynamic_cast` не может привести, то он падает в RE, в отличие от `static_cast` (СЕ). Правда, если классы вообще не совместимы, то будет СЕ. То есть RE в случаях сложной, запутанной иерархии. Если кастуем указатели, то вернется `nullptr`. В то же время `reinterpret_cast` просто будет считать часть сына от мамы за папу.

## 10.10 Virtual destructor

Когда мы привели объект какого-то типа к объекту типа его предка, то при удалении возникает проблема: при вызове деструктора вызывается деструктор старшего типа, то есть велика вероятность, что мы не все удалим. Сделав деструктор виртуальным, будет вызываться нужный деструктор. Также это самый простой способ сделать тип полиморфным.

\*\*\*

## 10.11 Pure virtual functions and abstract classes

Допустим мы пишем геометрию фигур и определяем класс `shape` – общий для всех фигур. У каждого наследника должен быть метод `area()` – посчитать площадь. У каждой отдельной фигуры данная функция понятна. А у `shape`? Нет. Её не понятно как определять.

```
1  virtual double area() = 0; // pure virtual function
```



Хоть одна чисто виртуальная функция  $\Rightarrow$  нельзя создавать объекты данного класса. (!Но можно его наследников!)

## 11 Templates — шаблоны

Все понимают, что и для чего это такое: позволяет писать обобщенные функции (классы).

### 11.1 Синтаксис

```
1  template<typename T>
2  T max(const T& a, const T& b) {
3      return a > b ? a : b;
4  }
```

```
1  template<typename T>
2  class vector {
3      ...
4  };
```

### 11.2 Typedef

```
1  template<typename T>
2  using strmap = map<T, std::string>
```

### 11.3 Перегрузка шаблонной функции

Если есть разные версии шаблонной функции, то работает перегрузка.

```
1  template<typename U, typename V>
2  void f(U x, V y); // назовем это функция 1
3
4  template<typename U>
5  void f(U x); // функция 2
6
7  void f(int x, double y); // функция 3
```

- Частное важнее общего;
- Чем меньше шаблонов, тем лучше, но каст хуже.

```
1  f(1, 1); // вторая версия из-за первого пункта
2  f(1.0, 1); // первая из-за второго пункта (без первой и второй будет СЕ)
```

```

1 // вместо третьей
2 template<...>
3 void f(T* x, T* y); // ...

```

Если у нас только

```

1 template<typename T>
2 void f();
3
4 f(); // CE

```

Можно делать шаблонный тип по умолчанию.

## 11.4 Специализация шаблонов

```

1 template<typename T>
2 class Class {
3
4 };
5
6
7 template<
8 class Class <int> {
9
10 };
11
12 Class<char> c; // первого типа
13 Class<int> s // второго типа

```

### 11.4.1 Частичная специализация

```

1 template<typename T>
2 class Class<int*> {};

```

Все это компилятор делает на этапе компиляции, генерируя отдельные функции. Частичная специализация возможна только при наличии общей!

### 11.4.2 Порядок специализаций

Важен порядок объявления специализаций (специализация подцепляется к ближайшей сверху общей)

```

1 template<typename T, typename U>
2 void f(T x, U y);
3
4 template<typename T>
5 void f(T x, T y);
6
7 template<
8 void f(int x, double y);

```

```

9
10 f(1, 0.1); // вызовется первый вариант
11
12 // если третий вариант поменять местами со вторым, то вызовется он (бывший третий)

```

### 11.4.3 template template parameters – шаблонный параметры

```

1  template<typename T, template<typename> class Container>
2  struct Stack {
3      Container<T> cont;
4  }
5
6  Stack<int, std::vector<int>> s;

```

Шаблонным параметром могут только базовые типы.

## 11.5 Метафункции

Допустим у нас есть функция от двух шаблонов

```

1  template<typename T, typename U>
2  void g() {
3      ...
4      // хотим if (T == U)
5  }
6
7
8  // хотим

```

Для этого:

```

1  template<typename T, typename U>
2  struct is_same {
3      static const bool value = false;
4  };
5
6  template<typename T>
7  struct is_same<T, T> { // специализация
8      static const bool value = true;
9  };
10
11 template<typename T>
12 struct remove_const<const T> {
13     using type = T;
14     // typedef T type;
15 };
16
17
18 template<typename T, typename U>
19 void g() {
20     ...
21     // хотим if (T == U)

```

```
22 if (is_same<T, U>::value){  
23     ...  
24 }
```

Все эти функции и другие есть в библиотеке `type_traits`

## 11.6 Неоднозначность

Когда у компилятора есть неоднозначность: тип или объект, то по умолчанию он выбирает объект. Чтобы обратиться к типу, то надо писать `typename`  
Чтобы убрать

...

## 11.7 Шаблоны с переменным количеством элементов

**P.S.:** Наименее предпочтительнее при выборе, что вызывать (так как описывают самый общий случай)

### 11.7.1 Синтаксис самого общего случая

```
1 template<typename... Args> // Args это название  
2 void f(Args... args) { // при компиляции это расшифровывается в список через запятую  
3     g(args...)  
4 }
```

### 11.7.2 Пример

```
1  
2 void print() {}  
3  
4 template<typename Head, typename... Tail>  
5 void print(const Head& head, const Tail& tail) {  
6     std::cout << head << " ";  
7     print(tail...);  
8 }
```

### 11.7.3 Переменное количество аргументов заданного типа

```
1 // тут должна быть реализация gcd для двух параметров, чтобы выходить из "рекурсии"  
2  
3 template<int N1, int... Nn>  
4 constexpr int gcd() {  
5     return gcd<N1, gcd<Nn...>()>();  
6 }
```

### 11.7.4 Количество переданных аргументов

```

1  void print() {}
2
3  template<typename Head, typename... Tail>
4  void print(const Head& head, const& Tail& tail) {
5      sizeof...(Tail)
6      std::cout << head << " ";
7      print(tail...);
8  }

```

## 12 Sequence containers

Vector, deque, list, forward\_list

### 12.1 Vector

#### 12.1.1 Объявление и прочие операции

```

1  vector<int> w;
2  vector<int> v(10); // в скобках первоначальный размер, в случае интов там будут лежать нули, то есть
                     // вызывается конструктор по умолчанию для каждого элемента
3  // если нет конструктора по умолчанию, то надо вторым параметром указывать значение
4  // для вектора векторов:
5  vector<vector<int>> v(100, vector<int>(100));
6
7  v.push_back(5); // добавление в конец
8  v.pop_back(); // удаление из конца; если из пустого, то UB!
9  v[1] = 3; // всё как с массивом
10 v.at(2) = 5; // разница от [] в том, что при обращении к элементу вне памяти появляется исключение. Минус
    в том, что там прописан if и это жрет время
11
12 v.size(); // сколько элементов реально лежит элементов
13 v.resize(20); // изменение размера с дополнением элементов
14 v.capacity() // сколько памяти выделено
15 v.reserve() // выделение доп памяти
16
17 // pop_back() не уменьшает память!!
18 // поэтому в C++ 11 появился
19 v.shrink_to_fit(); // сжимает память до количества элементов в векторе
20
21 // !!!! sizeof(v) не зависит от количества элементов в векторе
22
23 v.back() // ссылка на последний элемент
24 v.front() // ссылка на первый элемент
25 // !!! если пустой, то UB
26
27 // более быстрый и эффективный метод, чем push_back()
28 v.emplace_back(); // см реализацию

```

### 12.1.2 Реализация

```

1  template<typename T>
2  class vector {
3      private:
4          T* arr;
5          size_t sz;
6          size_t cp;
7
8      public:
9          vector(size_t n, const T& valueT()): sz(n), cp(n) {
10             arr = new char[n * sizeof(t)];
11             for (size_t i = 0; i < n; ++i) {
12                 // placement new
13                 new(arr + i) T(value);
14             }
15         }
16
17         void push_back(const T& value) {
18             if (sz == cp) {
19                 // reallocate()
20             }
21             // placement new
22             new(arr + sz) T(value);
23         }
24
25         void pop_back() {
26             --sz;
27             arr[sz].~T();
28         }
29     };
30
31     template<typename... Args> // переменное количество аргументов
32     void emplace_back(const Args&... args) {
33         ...
34     }

```

### 12.1.3 Вектор bool

Особую реализацию имеет вектор булей (записывает по 8, чтобы один буль занимал бит, а не байт).

```

1  template<>
2  class vector<bool> {
3      // ...
4      private:
5          struct BoolProxy {
6              int& b;
7
8              BoolProxy operator =(bool b) {
9                  // bx = b; по сути, но мы работаем с интами
10                 // взять нужный бит и изменить его
11                 return *this;
12             }
13         };
14     };
15

```

```

16     BoolProxy operator [] (size_t n) {
17         return BoolProxy(arr[n], n%8);
18     }
19 };

```

## 12.2 Deque

От вектора отличается следующим:

- наличием `push_front(value)`
- наличием `pop_front()`
- отсутствием `capacity()`
- отсутствуют `reserve()`

## 12.3 List

В плюсах это двусвязный список.

...

# 13 Associative containers

Примеры: `map`, `unordered_map`

## 13.1 Map

Красно-черное дерево (стандартом не уточнено какое дерево, по сути любое подойдет, главное чтобы основные операции за логарифм были).

Хранит пары “ключ-значение”, поэтому ключи должны быть сравнимы (они хранятся в узлах дерева)

### 13.1.1 Некоторые операции

```

1  std::map<std::string, int> m;
2
3  m["asasf"] = 2;
4  m["asdas"] = 5;

```

Из выше написанного можно сделать вывод, что квадратные скобочки создают дефолтный элемент ⇒ если мэп константный, то квадратные скобочки не работают.

Поэтому существует метод `at()`, не создающий ничего (а следовательно и не меняющий). Ради эффективности есть метод `insert(std::pair<“ключ”, “значение”>)`

```

1  m.insert(std::make_pair("aaaa", 7));
2  // с C++ 11 можно вместо make_pair(...) писать {value1, value2}

```

Для еще большей эффективности с C++ 11 есть `emplace(key, value)`, который сам создает пару.

```
1 m.find(key); // возвращает итератор на элемент с данным ключом
2 // если не нашла, то возвращает end()
```

## 13.2 Unordered map

(с C++ 11)

Всё то же, что в `map`, только операции за  $O(1)$ , а не за  $O(\log n)$ , потому что реализован на хеш-таблице.

## 14 Exceptions – Исключения

Исключения появляются только тогда, когда

### 14.1 Общий синтаксис

```
1 try {
2     // код, который потенциально может генерировать ошибку (исключение)
3     // если он генерирует, то мы приостанавливаем данный код и идем в секцию catch
4 } catch (const std::out_of_range& err) { // в скобках указывается тип ошибки
5     // код только в случае данной ошибки
6     // если случилось то, что catch не отловил, то программа валиться
7 }
```

### 14.2 Как кидать исключение

```
1
2 int f() {
3     throw std::out_of_range("текстовое описание"); // лежит в библиотеке stdexcept
4     // throw – это оператор
5     return 0;
6 }
```

**P.S.:** бросать можно любой объект

Ещё два оператора, которые бросают исключения: `new` (если не хватило памяти) и `dynamic_cast`

### 14.3 Дополнительный синтаксис

Ловить можно сколько угодно раз, приведение типов нет, за исключением приведение наследников к предку (например, указав ловить `exception`, общий тип исключения, можно поймать `out_of_range`, так как он является наследником)

```
1 try {
2     // ...
3 } catch (const std::out_of_range& err) {
4     // ...
5 } catch (int x) {
```



```

6 // ...
7 } catch (...) {} // поймать что угодно

```

**P.S.:** Выполняется только один catch!!! При этом выполняется именно тот catch, который подойдет первым. (Например, если ловить `exception` до `out_of_range`, то второй не паймается никогда.

Если мы из одного catch'а хотим попасть в другой, не создавая копии. Тогда внутри catch достаточно просто написать `throw`;

В процессе выбрасывания исключения происходит два копирования: объекта (который выбрасывается, например, строка) из стека в место памяти для исключений, а потом в catch. Можно избавиться от одного копирования, если ловить ссылку.

```

1 } catch (const std::exception& err)

```

## 14.4 Метод what()

У всех исключений есть метод `what()`, который возвращает строку, переданную в конструктор исключения.

```

1 throw std::out_of_range("abcde");
2 ...
3 } catch (const std::exception& err) {
4     std::cout << err.what();
5 }

```

## 14.5 Memory leaks caused by exceptions

```

1 int g() {
2     int* p = new int;
3     // ...
4     // ...
5     int x = f();
6
7     delete p;
8     return x * 2;
9 }

```

Допустим, в функции выше вызвалось исключение до команды `delete`. Тогда мы не освободили память. Данный пример простой и такое легко отследить, но в реальности выражения могут быть в разы сложнее и прерывание функции в середине может привести много проблем.

Решить данную проблему помогают умные указатели (подробнее можно почитать дальше):

```

1 int g() {
2     // RAII – Resource Acquisition Is Initialization
3     std::shared_ptr<int> p(new int);
4     // ...
5     // ...
6
7     int x = f();

```

```

8 |     return x * 2;
9 | }

```

### 14.5.1 Исключения в конструкторах

```

1 | struct MyStruct {
2 |     int* p;
3 |
4 |     MyStruct() {
5 |         p = new int;
6 |         f(p);
7 |     }
8 |
9 |     MyStruct(const MyStruct&) = delete;
10 |    MyStruct& operator =(const MyStruct&) = delete;
11 |
12 |    ~MyStruct() {
13 |        delete p;
14 |    }
15 | };

```

Допустим `f(p)` вызывает исключение. В таком случае деструктор не будет вызываться (потому что конструктор не завершил свою работу), но деструкторы полей вызваны будут! (потому что их создание завершено)

**P.S.:** опять же, в данной конкретной ситуации нас бы спасли умные указатели.

### 14.5.2 Исключения в деструкторах

```

1 | struct VeryBad() {
2 |
3 |     int x;
4 |     VeryBad(int x): x(x) {}
5 |
6 |     ~VeryBad() {
7 |         f(x);
8 |     }
9 |
10 | };

```

Если в деструкторе потенциально может появиться исключение, то это **совсем плохо**.

Например:

```

1 | int g() {
2 |     VeryBad vb = 1;
3 |
4 |     throw 1;
5 |     return 0;
6 | }
7 |
8 | int main() {
9 |     try {

```

```

10     g();
11 } catch (int x) {
12     std::cout << x;
13 }
14 }

```

В данном примере `g()` начинает кидать исключения. Для этого вызываются деструкторы всех созданных в `g()` объектов, а там деструктор `vb` вызывает ещё одно исключение. В таком случае, сразу вызывается функция `terminate` и программа сразу крашется.

В таком случае нам может помочь встроенная функция `std::uncaught_exception()`:

```

1 struct VeryBad() {
2
3     int x;
4     VeryBad(int x): x(x) {}
5
6     ~VeryBad() {
7         if (!std::uncaught_exception()) {
8             f(x);
9         }
10    }
11
12 };

```

## 14.6 Спецификация исключений

(с C++11)

Есть возможность указать в сигнатуре, кидает ли потенциально функция исключение.

```

1 void h(const std::string& s) noexcept { // данная функция не кидает исключения
2
3 }

```

**P.S.:** к сожалению, `noexcept` — это лишь обещание, а не проверка компилятором или ещё что-то. Если функция с `noexcept` кидает исключение, то это сразу `terminate` и никак не отловить.

### 14.6.1 Оператор `noexcept`

Допустим, мы хотим сделать функцию, которая лишь в некоторых случаях не кидает исключения.

```

1 noexcept(f()) // true если f() и всё, что внутри noexcept, false иначе

```

**P.S.:** само `f()` не вычисляется!!!

Таким образом можно писать:

```

1 void h(const std::string& s) noexcept(noexcept(f(x))) {
2     // внутри это оператор, а вне – спецификатор
3     f(x)
4 }

```

## 14.7 function-try block

...

## 14.8 Finally in Try-Catch

Этого в плюсах нет, страдай :)

# 15 Iterators – итераторы

Хотим пройтись по мапе пар и вывести элементы. Есть два основных метода, как это сделать.

```

1  void f(const std::map<int, int>& m) {
2      // 1 метод — range based for / foreach
3
4      // since C++11
5      for (std::pair<int, int> p : m) { // тут еще можно убрать копирование заменив на ссылку и т.п.
6          std::cout << p.first << " " << p.second << std::endl;
7      }
8
9      // 2 метод — константный итератор
10     // в данном случае именно константный, потому что мап константный
11     for (std::map<int, int>::const_iterator it = m.begin(); it != m.cend(); ++it) {
12         std::cout << it->first << " " << it->second << std::endl;
13     }
14 }
```

## 15.1 Categories of iterators

Любой итератор позволяет разыменовывать себя.

### 15.1.1 Input

Это такие, которые позволяют один раз пройтись по себе.

### 15.1.2 Forward

Умеют только вперед ++. Содержатся в `forward_list` и `unordered_map`

### 15.1.3 Bidirectional

Умеют в обе стороны ++ и --. Содержатся в `list`, `map`, `set`

### 15.1.4 Random access

Умеют какой угодно по номеру ++, --, <, >, +c, -c. Содержатся в `vector`, `deque`

### 15.1.5 Contiguous

(с C++11)

Это самые крутые итераторы - как Random Access, только гарантируют, что в памяти объекты в одном месте.

## 15.2 Iterator Traits

```

1  template<typename> Iter
2  struct iterator_traits {
3      using value_type = //...
4      using iterator_category = //...
5  };

```

## 15.3 Distance and Advance

Речь пойдет о встроенных функциях `std::distance` и `std::advance`

```

1
2  template<bool b>
3  void advance_impl(Iterator it, int n) {
4      for (int i = 0; i < n; ++i) {
5          ++it;
6      }
7  }
8
9  template<>
10 void advance_impl<true>(Iterator it, int n) {
11     it += n;
12 }
13
14 template<typename Iterator>
15 void advance(Iterator it, int n) {
16     // прибавляем к итератору n, причем грамотно: если random access, то за O(1) иначе за O(n)
17     advance_impl<std::is_same<typename std::iterator_traits<Iterator>::iterator_category,
18         std::random_access_iterator_tag>::value>(it, n);
19 }

```

## 15.4 Standart algorithms

В библиотеке `algorithms` полно алгоритмов, которые принимают диапазон

```

1  Iterator binary_search(Iterator begin, Iterator end, const ValueType& value);

```

## 15.5 Output Iterators

Данные итераторы позволяют безопасно к ним обращаться, типа такого:

```

1  template<typename InputIterator, typename OutputIterator>
2  void copy(InputIterator begin, InputIterator end, OutputIterator output) {
3      while (begin != end) {
4          *output = *begin;
5          ++output, ++begin;
6      }
7  }

```

## 15.6 Insert, Back\_insert and Front\_insert iterator

Позволяют вставлять в контейнеры (каждый существует у своего множества контейнеров, когда

...

## 15.7 Iterators and references invalidation

Рассмотрим код:

```

1  std::vector<int> v(10);
2  std::vector<int>::iterator it = v.begin();
3  for (int i = 0; i < 100; ++i) {
4      v.push_back(i);
5  }
6  *it = 5;

```

Данный код вызывает UB, потому что вектор изменит свой размер в ходе цикла, а значит итератор будет указывать не понятно куда (в то же место памяти, но мы не знаем, что там). То же самое с указателями или ссылками. Если вектор был изменен, то итераторы, указатели, ссылки инвалидируются.

**P.S.:** с листом, например, это не так, из-за принципа работы `list` - он ничего не куда не перекладывает.

Итого:

- `vector`: invalidates
- `list`: doesn't invalidate
- `deque`: invalidates iterators but not references.
- `map`: doesn't invalidate
- `unordered_map`: invalidates iterators but not references.

В `unordered_map` надо понимать, что итераторы инвалидируют, но при этом они будут указывать на действительный элемент (просто при изменении данного контейнера, может поменяться порядок обхода, если случится `rehash()`, чего можно избежать, например, выделив заранее кучу места).

## 15.8 Auto, decltype

(с C++11)

Чтобы не писать каждый раз необходимый тип, можно использовать `auto`, также как `const auto` и `const auto&`

```

1  int a = 5;
2  // хотим
3  auto b = 'a';
4  // но не типа char, а такого же типа, что и a
5  decltype(a) b = 'a'; // наше решение

```

`decltype(expression)` .... — в скобках expression

**P.S.:** можно также объявлять и функции

## 16 Move semantics и RVALUE ссылки

### 16.1 Что это такое и мотивация

```
1 std::vector<std::vector<int>>> v;
2 v.push_back(std::vector<int>());
```

В данном коде будет лишнее копирование. Поэтому в **C++11** было изменено ядро кода. Добавили move semantics. Это позволяет в нужные моменты не выполнять лишние копирования.

### 16.2 Magic function std::move()

Данная функция лежит в библиотеке `utility`

Когда хотим скопировать, надо писать данную функцию. Например `swap()`:

```
1 template<typename T>
2 void swap(T& a, T&b) {
3     T t = std::move(a);
4     a = std::move(b);
5     b = std::move(t);
6 }
```

Таким образом, как нужно писать вектор:

```
1 vector& operator=(vector<T>&& v) {
2     this->~vector();
3     sz = v.sz;
4     cp = v.cp;
5     arr = v.arr;
6
7     v.arr = nullptr;
8 }
9
10 vector(vector<T>& v): sz(v.sz), cp(v.cp), arr(v.arr) {
11     v.arr = nullptr;
12 }
```

И тут пора поговорить про Rvalue ссылки. Это специальный тип, который показывает, что надо не копировать, а перемещать. `std::move()` соответственно кастует (если говорить упрощенно, для первого знакомства) к rvalue ссылке. Если не определено для `&&`, то происходит каст к `&` и будут копирования. Поэтому появилось правило пяти, которое расширяет правило трех на следующие правила:

1. мув конструктор
2. мув оператор присваивания