

# 1 Клуб фанатов Manowar

1. Каждая секция - отдельная тема у Сорокина ([sorokin.github.io/cpp-course](http://sorokin.github.io/cpp-course)).
2. Для списков используйте `\begin{enumerate}` или `\begin{itemize}`
3. Очень помогает latex cheatsheet.
4. Первые пять билетов пропущены - там асм и прочая дичь. Пока что в приоритете билеты по плюсам.
5. Может попасться всё до 19 билета включительно.
6. Писать максимально подробно по каждому пункту - мало ли, что могут спросить.
7. Я заменил "Билет" на "Лекции". Так корректнее (билетов на экзамене как таковых нет - будут рандомные вопросы по лекциям. При этом лекции 1-5 попадают редко.)

## 2 Лекция 6

### 2.1 Структуры

С помощью структур можно создавать пользовательские типы. По дефолту все поля структуры публичны, в то время как поля классов по дефолту приватны.

Доступ к полям структур осуществляется через оператор `.`, для указателей на структуры используется оператор `->`.

```
struct point {  
    double x;  
    double y;  
};  
...  
point p = {11.4514, 810.931};  
p.x = -19.19;  
point * pp = new point;  
pp->y = 4545;
```

Инициализировать структуры можно, прописав фигурные скобки.

Методы — функции, определяемые в полях структур (non C-style). Они реализованы с неявным параметром `this`, который является указателем на текущий экземпляр структуры. Объявление и определение методов:

```
struct point {  
    // declaration  
    void shift(double x, double y);  
    double x;  
    double y;  
};  
...  
// definition  
void point::shift(double x, double y) {...}
```

Определение структуры не генерирует никакого кода на ассемблере. Структуры существуют только на момент компиляции, они определяют то, как данные адресуются в памяти и располагаются, но после того, как код скомпилировался, никакой информации о структурах уже нет. Конструкторы — методы для инициализации структур.

```
point() {x = y= 0;}
point(double x, double y) {
    this->x = x;
    this->y = y;
}
...
// empty constructor call
point p1;
// calling constructor with 2 args
point p2(1, 2);
```

Список инициализации позволяет проинициализировать поле до входа в тело конструктора. Инициализация полей в списке инициализации происходит в порядке объявления полей. То есть при написании `y(0)`, `x(y)` произойдёт не то, что ожидалось.

```
point() : x(0), y(0) {}
point(double x, double y) : x(x), y(y) {}
```

В функциях можно определить значения по умолчанию.

```
point(double x = 0, double y = 0) : x(x), y(y) {}
...
point p1; // x = 0, y = 0
point p2(2); // x = 2, y = 0
point p3(3, 4); // x = 3, y = 4
```

Конструктор одного параметра задаёт неявное пользовательское преобразование. Запретить это можно при помощи ключевого слова `explicit`.

```
explicit point(double x = 0, double y = 0) : x(x), y(y) {}
...
point p = 2; // error
```

Конструктор по умолчанию генерируется компилятором, если он пользовательски не задан.

```
struct segment {
    segment(point p1, point p2) : p1(p1), p2(p2) {}
    point p1;
    point p2;
};
...
segment s; // error
```

Деструктор — метод, вызывающийся при удалении структуры и по умолчанию генерируется компилятором. У него может быть только 0 аргументов, и он всегда один на одну структуру. Объявляется как `<struct name>`.

Конструкторы и деструктор позволяют чётко определять время жизни структуры. Обращение к некоторым значениям до вызова конструктора или после вызова деструктора приводит к `undefined behavior`.

## 2.2 Указатели, адреса, массивы и их друзья

Память компьютера (спасибо SKKV'у) - это последовательность ячеек, пронумерованных от 0 до  $2^n - 1$ . Всё, что в ней расположено, имеет свой адрес. Тип данных, хранящий адрес на объект типа  $t$  называется указателем на  $t$ :

```
int a = 666;
int *ptr = &a; // *a = 666, унарный & - оператор взятия адреса в памяти.
```

Например, если наша переменная лежала в участке памяти, **первая** ячейка которого имеет номер 1337228, указатель будет хранить число 1337228. Из коммента к примеру видно, что унарный \* - это оператор разыменовывания. Кроме того, можно разыменовывать указатель и делать присваивание:

```
char ch = 'a', ch2 = 'q';
char *ptr = &ch;
char *ptr2 = &ch2;
*ptr = 'b'; // OK, ch = 'b'.
*ptr = *ptr2; // OK, ch = 'q', ch2 = 'q'
int* fail = &ch2; // CE, нельзя присваивать int* char*
int* rr = new int(8);
delete rr;
*rr = 111; // ОЖВП
```

Массивы - это, по сути своей, последовательность объектов. Они расположены в соседствующих ячейках. В качестве синтаксического сахара существует оператор [].

**NB:** размер статического массива определяется на этапе компиляции.

```
int arr[10];
int* qq = arr; // OK, массив, по сути, это и есть указатель.
arr[0] = 12, arr[1] = -15, arr[2] = 666;
printf("%d", *arr); // напечатает первый элемент, указатель хранит именно его.
*(arr + 2) = 20; // arr[2] = 20.
arr[1] = 100;
*(arr + 1) = 100; // это эквивалентный код.
arr[-1] = 1349; // i am not in danger, i AM the danger.
// RTE такое не выкинет, осторожно!
arr[5] = 88888;
int *ptr = &arr[5]; // OK, указатели могут ссылаться на элементы массива.
*ptr = 102; // OK, arr[5] = 102
```

Из примера выше: в арифметике указателей не требуется домножать на  $sizeof(type)$ . Инкремент указателя на  $int$ , например, увеличит адрес на 4, а не на 1, так как  $sizeof(int) = 4$ . Декремент работает аналогично, но уменьшает адрес. Примеры с арифметикой:

```
int arr[10];
int *ptr = arr;
*(++ptr) = 1214; // OK, arr[1] = 1214, внимание на приоритет операций.
int *ptr2 = &arr[8];
printf("%d", ptr2 - ptr); // OK, выведется 7.
auto bye = ptr + ptr2; // CE, складывать нельзя.
```

```
for (int *i = &arr[0]; i<&arr[10]; ++i) printf("%d ", *i); // ОК, выведет arr.
```

Ещё есть тип *\*void*, он юзается для указателей на ячейку памяти, тип которой неизвестен.

Пример юзания - передача адреса из одной части программы в другую.

Указатели на функции:

```
int foo(){
    return 102;
}
double square(double a){
    return a*a;
}
int area(int a, int b){
    return a*b;
}
int (*ptr)() = foo; // ОК. Внимание - тут неявная конвертация
// NB: если бы у нас была ф-ия сортировки с аргументом - указатель на функцию
// компаратора (int* arr, bool (*comparator)(int, int))
// и компаратор cmp где-нибудь сверху, можно было бы написать sort(arr, cmp);
int (*ptr2)() = square; // СЕ, разные типы возврата
double(*ptr2)() = square; // СЕ, не те аргументы
double(*ptr2)(double) = square; // ОК
int (*ptr3)(int, int) = area; // ОК
// вызов функций:
std::cout << (*ptr)() << "\n"; // 102
std::cout << (*ptr2)(13.22) << "\n"; // 174.168
std::cout << (*ptr3)(12, 22) << "\n"; // 264
```

Такой уродский синтаксис обусловлен приоритетом операций. В противном случае код `int *ptr()` будет прочтён как предварительное объявление функции *ptr()*, которая ничего не принимает и возвращает указатель на *int*.

И немного о ссылках: это, по сути, автоматически разыменовываемый константный указатель.

```
int t = 10;
int &link = t;
link = 666; // t = 666
int a = 124124;
link = &a; // СЕ
```

## 3 Лекции 7/8

### 3.1 Процесс компиляции программ

#### 1. Препроцессинг.

На этой стадии происходит работа с директивами препроцессора. Например, `#include`, `#define`, `#ifndef`. Хэдеры, включённые через `#include` проходят препроцессинг рекурсивно. При помощи флага `-E` можно получить файл без компиляции, но с препроцессингом. `g++ -E A.cpp -o A.i` - здесь `A.i` это `.cpp` файл после препроцессинга.

#### 2. Получение ассемблерного кода.

Здесь плюсовый код превращается в ассемблерный (но ещё не машинный). При помощи флага `-S` можно получить ассемблерный код: `g++ -S A.cpp -o A.s`

#### 3. Получение машинного кода.

Здесь тексты на языке ассемблера превращаются в объектные файлы. Объектный файл содержит кусок машинного кода, который ещё не связан с остальными кусками кода в единую программу. Простейший пример - определение полей класса в одном файле, а объявление - в другом. Именно для объединения их и существует следующая стадия. Получить машинный код с помощью асма AT&T можно так: `as A.s -o A.o`

#### 4. Линковка.

Здесь все объектные файлы и статические либы связываются в единый исполняемый файл, который мы и используем. Для линковки используется таблица символов. Это генерируемая компилятором структура данных, хранящая имена функций, объектов итд, где каждому идентификатору (символу) соотносятся его тип и область видимости. В ней также хранятся адреса ссылок на данные и прочее в других объектных файлах. Благодаря этой таблице линковщик строит связи между данными в различных объектных файлах, собирая из них единый исполняемый файл / библиотеку.

**NB:** Единицей трансляции называется подаваемый на вход компилятору исходник со всеми включёнными в него файлами. Судя по англовики, это то, что получается после первой стадии.

### 3.2 declaration/definition

Объявление вводит переменную/функцию в область видимости, устанавливает её тип и, опционально, инициализирует её. Пока имя не объявлено, его использовать, очевидно, нельзя. Объявление, которое полностью описывает объявленную сущность (переменная/функция/etc) называется определением.

```
double square(double); // declaration функции
extern int tmp; // declaration (defined by garbage value)
int def = 98; // declaration (and definition)
int square(double); // очевидный CE.

double square(double k){
    return k*k;
} // definition of declared "square"

// cycling:

double f(int a){
    return g(a-1) - 44;
}
```

```
double g(int a){
    if (a > 1) return f(a-29) + 151;
    return 9;
} // цикл определений, чтобы спастись, нужно заранее объявить функции
// struct def/decl:

struct point;

struct point {int x; int y;};

struct cycle{
    cycle a; // CE
    cycle *a; // OK
};
```

### 3.3 static, headers, header-guard (карты, деньги, два ствола)

Модификатор `static` работает как в жаве - это **общее** поле для всех экземпляров класса. Можно обратиться извне (если публичный) при помощи двух двоеточий. Использование `static` по отношению к члену класса гарантирует, что в программе будет ровно одна копия его значения (а не по копии на каждый объект). Статические переменные в функциях создаются и инициализируются только один раз, а после выхода из области видимости - не сгорают. Тем не менее, к ним можно обратиться только из их области видимости. Статические методы классов не принимают `this` и могут обращаться только к `static` полям.

**NB:** Статические функции и переменные в **глобальной видимости** не проходят внешнюю линковку - из сторонних файлов их не увидеть.

```
void foo(){
    static int a = 1;
    ++a;
    printf("%d\n", a);
}

int main(){
    foo(); // 2
    foo(); // 3
    foo(); // 4
    a = 666; // CE, нет в области видимости.
}
```

Заголовки позволяют программисту контролировать определения и объединения. В хэдерах записывают объявления, а затем их можно помещать при помощи `#include`. В C++ принята следующая практика (на примере): в `sample.h` файле собраны объявления, а `sample.cpp` - инcludes хедер и определяет поля. Например, в `calc.h` объявлена функция `ln`, а в `.cpp` - эта функция реализуется. Позже в другом файле сделать `#include "sample.h"` и пользоваться той же функцией `ln`.

Однако, есть одна проблема. Директивка `#include` тупо копирует содержимое другого файла в исходник. Поэтому может возникнуть такое:

```
// A.cpp
struct A{
```

```
        void foo();
    };
    // B.cpp
    #include "A.cpp"
    struct B{};
    // C.cpp
    #include "A.cpp"
    struct C{};
```

После препроцессинга А будет определён в двух единицах трансляции, что может привести к UB. Для защиты от такого существует **include-guard**, который не позволяет заинклудить больше 1 раза. Он весьма прост.

```
// foo.h
#ifndef FOO_H
#define FOO_H
struct foo{
    int a;
    void bar();
};
#endif
```

Здесь во время препроцессинга будет проводиться проверка - определён ли идентификатор макроса FOO\_H. Если он не определён - он определяется, а содержимое успешно присоединяется к файлу. Если попробовать присоединить второй раз, то идентификатор FOO\_H уже будет определён и if не сработает, препроцессор пропустит весь код до #endif и таким образом избежит второго определения структуры.

Альтернатива: тупо вставить #pragma once в начало хедера.

## 4 Лекции 9/10

### 4.1 Классы, функции члены-класса, this

Классы в C++ — структура с методами, конструктором и деструктором.

Функции члены-класса — методы.

Экземпляр класса — объект.

```
struct IntArray {
    explicit IntArray(size_t size) : size(size), data(new int[size]) {
        for (size_t i = 0; i < size; ++i) {
            data[i] = 0;
        }
    }
    ~IntArray() {
        delete[] data;
    }
    int & get(size_t i) {
        return data[i];
    }
}
```

```
    size_t size;
    int *data;
};
...
IntArray a(10); // объект
IntArray b = {20, new int[20]}; // ошибка, для инициализации теперь есть конструктор
IntArray *c = new IntArray(10); // указатель на объект в динамической памяти
```

`this` — константный указатель(?), являющийся `rvalue`, присутствующий неявно в каждом методе класса. Вместо `data` можно использовать `this->data`.

## 4.2 Модификаторы доступа

Модификаторы доступа — ключевые слова, нужные для изменения возможности доступа к данным в классе.

Существуют 3 таких модификатора: `public`, `private` и `protected`. Первый означает, что у внешних функций есть доступ к данным класса, второй — наоборот не даёт доступа никому, а последний — как `private`, только доступ есть у наследников.

```
struct IntArray {
    ... // <- здесь находятся public данные
private:
    size_t size;
    int *data;
};
```

Для структур (`struct`) по умолчанию стоит `public`, а для классов (`class`) — `private`.

## 4.3 Ссылки

Ссылки — специфические константные указатели.

Сравнение ссылок и указателей:

1) ссылка не может быть неинициализированной.

```
int &l; // СЕ
```

2) у ссылки нет нулевого значения.

```
int &l = 0; // СЕ
```

3) ссылку нельзя переинициализировать.

```
int &l = a; l = b; // на самом деле произошло a = b
```

4) нельзя получить адрес ссылок и ссылку на ссылку.

```
int &l = a; int *pl = &l; // получили ссылку на a
```

```
int &&l = l; // СЕ
```

5) нельзя создавать массив ссылок (так как ссылки обязательно нужно инициализировать)

6) нет арифметики для ссылок

Важный момент: указатели и ссылки могут указывать только на `lvalue`.

## 4.4 Чем отличается структура от класса?

Поля структуры по умолчанию публичные, у класса поля по умолчанию приватные.



## 4.5 Конструкторы и деструкторы

Функция-поле, имя которой совпадает с именем класса, используемая для инициализации объектов класса, называется **конструктором**. Есть и обратная функция, которая вызывается неявно, по завершении жизни объекта, об этом чуть ниже. Такая функция называется **деструктором**. Он не может принимать какие-либо аргументы.

```
struct PVector{
    int* data;
    PVector(size_t a){
        if (a <= 0)
            throw std::runtime_error("incorrect size");
        data = (int*) malloc(a*sizeof(int));
    } // конструктор
    void set(size_t ind, int a){
        data[ind] = a;
    }
    ~PVector(){
        free(data);
    } // деструктор, автоматически освобождает память
}

int main(){
    PVector a; // CE
    PVector a(12); // OK
    PVector a(-2); // RTE, которую мы устроили сами
    PVector b = PVector(777); // OK
}
```

Про `explicit` конструкторы написано в 6й лекции.

## 4.6 storage classes & object lifetime

Существует три класса памяти.

1. Автоматическая память - переменные/параметры функций размещаются в ней (то есть в стеке). Эта память выделяется при вызове функции и освобождается при возвращении управления туда, где её вызвали.
2. Статическая память - там лежат переменные из глобальной области видимости, а также `static`-переменные в функциях и классах. Линковщик выделяет эту память до запуска программы.
3. Динамическая память - ну тут и так понятно. Там хранятся созданные через `new` объекты. Удаление/выделение на плечах программиста.

Время жизни объекта определяется областью видимости и классом памяти.

- Локальные (они же автоматические) объекты рождаются в момент их определения и умирают на выходе из области видимости.
- Объекты в неймспейсах, а также статика создаются до `main()` и уничтожаются после `main()`.
- Локальные статические объекты (например, статика в функциях) создаются в момент определения и уничтожаются после `main()`.

- Объекты в динамической памяти создаются во время **new**, уничтожаются во время **delete**.
- Временные объекты (это то, что появляется, например, в `bigint a = b + c`) создаются подвыражением и уничтожаются по завершении полного выражения (полное не является подвыражением другого выражения). Если временная переменная связана с какой-то ссылкой - она будет жить, пока жива ссылка. Пример:

```
std::vector<int> a = {1, 2, 3};
void get(int q) return a[q];

void func(){
    const string &link = get(0);
    string qq = get(2);
}
// временная строка из get(0) живёт до конца func.
// временная строка из get(2) уничтожается после инициализации qq.
```

(Перегрузку операторов пропустил - и так понятно, как в джаве.)

## 4.7 Специальные функции-члены класса

### 4.7.1 Конструктор по умолчанию

Если в классе **не определён ни один конструктор**, то тогда будет создан конструктор по умолчанию. Он не имеет тела и не принимает аргументов. Очевидно, что их можно перегружать.

### 4.7.2 Деструктор

Вызывается при уничтожении объекта, не может принимать аргументов, по умолчанию ничего не делает. По окончании работы деструктора будут неявно вызваны деструкторы всех не-статических членов класса. Деструкторы нельзя перегрузить.

### 4.7.3 Конструктор копирования

Рассмотрим `PVector` из 4.5. Что будет, если мы создадим один вектор и присвоим его другому? Оба указателя *data* будут ссылаться на одну и ту же область памяти! А это значит, что если поменять какое-нибудь значение в  $v_2$  при помощи *set*, оно поменяется и в  $v_1$ . А ещё есть риск двойного удаления. Очень жаль.

Конструктор копирования по умолчанию сгенерируется, если он не определён в коде (а ещё - если в классе нет виртуальной дичи(11/12)). Как легко догадаться, дефолтный конструктор копирования делает тупейшее битовое копирование. Синтаксис конструктора копирования: `PVector (const PVector&)`. В случае с нашим `PVector` мы можем переопределить так - выделить память у нового объекта (то есть указатель на данные будет другим) и сделать уже туда тупой memcpy.

**НВ:** Можно вообще запретить копирование объектов - определить конструктор копирования и засунуть его в приватную область.

#### 4.7.4 Оператор присваивания

Вернёмся к `PVector`. Хотя код `PVector a = v2; PVector b(v1);` стал безопасен (здесь  $v_1, v_2$  это ранее созданные `PVector`), код `b = a;` всё ещё представляет угрозу. Здесь ещё и новая катастрофа - будет очевидная утечка (кто освободит память в `b`?) Дефолтный оператор присваивания работает по аналогии с дефолтным конструктором копирования. Его синтаксис: `PVector& operator=(const PVector& other)`. А чтобы пофиксить `PVector` потребуется не только аккуратно выделить память/скопировать, но и освободить старую память. В конце оператора присваивания (не обязательно `PVector`) следует сделать `return *this;` (внимание на то, что возвращает оператор).

#### 4.8 deleted functions

Выше было указано, как можно запретить копирование объектов. Начиная с C++11 есть более изящный метод, как это сделать:

```
struct foo{
    foo();
    foo(const foo& a) = delete; // запрещено копирование
    foo& operator=(const foo& a) = delete; // запрещено присваивание
    void *operator new(std::size_t) = delete;
    void *operator new[](std::size_t) = delete; // запрещён оператор new
    void bar(int) = delete; // запрещён автоматический каст
    void bar(double);
}
```

Кроме того, на удалённые функции нельзя создавать указатели. Всё отсеивается на этапе компиляции.

#### 4.9 new/delete/malloc/free

Общее у этих выражений - они выделяют/освобождают динамическую память соответственно. Отличия:

- `new` в случае неудачи кинет исключение, `malloc` вернёт `nullptr`.
- `new` вызывает конструктор и выделяет память (или только конструктор, если это placement `new`). `malloc` только выделяет память.
- аналогично: `delete` вызовет деструктор, `free` только освободит память.
- `new` берёт память из `free store`, `malloc` - из `heap`. По сути, это две различных области памяти. В одной то, что выделено `new`, в другой то, что выделено `malloc`. Это одна из причин, почему `fsanitize`, связанные с памятью, очень сильно горят на попытку сделать `free` к памяти, выделенной `new` и наоборот.

Отличие `new/delete` от `new[] / delete[]`:

`new` выделит память и вызовет конструктор для одного объекта, `delete` - освободит и позовёт деструктор для одного объекта соответственно.

`new[]` выделит память и вызовет конструкторы каждого объекта для массива объектов, `delete` - освободит и позовёт деструкторы для каждого элемента в массиве объектов соответственно.

UB в new/delete:

- new[], но delete.
- ptr = new[]; delete[] (ptr + c) (c - константа).
- (подробнее в 19): ptr = new derived[666]; delete static\_cast<base\*>(ptr);.

## 5 Лекции 11/12

### 5.1 Наследование

Наследование — механизм, позволяющий создавать производные классы, расширяя уже существующие.

```
struct Person {
    string name() const {
        return name_;
    }
    int age() const {
        return age_;
    }
private:
    string name_;
    int age_;
};

struct Student : Person {
    string university() const {
        return uni_;
    }
private:
    string uni_;
};
```

Класс-наследник будет обладать теми же полями и методами, что и у класса-родителя.

```
Student s;
s.age(); // не ошибка
```

Внутри объекта класса-наследника хранится экземпляр родительского класса.

Поля, добавленные в классе-наследника, инициализируются внутри конструктора класса-наследника, а унаследованные поля — внутри конструктора родителя. При создании объекта сначала вызывается конструктор родительского класса, а затем — конструктор производного класса. (Если конструктор родительского класса указан пользовательски, его нужно вызывать в списке инициализации конструктора производного класса).

```
Person(string &name, int age) : name_(name), age_(age) {}
...
Student(string &name, int age, string &uni) : Person(name, age), uni_(uni) {}
```

Если у **Person** есть конструктор без аргументов — его вызывать самостоятельно не нужно.

Аналогично работает деструктор: сначала вызывается у **Student**, а затем — у **Person**. Это делается автоматически и на порядок его выполнения никак нельзя повлиять.

Производные классы связаны со своими базовыми при помощи приведений. Для ссылок и указателей на объекты производного класса определены приведения к указателям и ссылкам базового класса.

```
Student s("Alex", 21, "Oxford");
Person &l = s;
Person *r = &s;
```

Обратное приведение не работает.

Объекты класса-наследника могут присваиваться объектам родительского класса.

```
Person p = s; // Person("Alex", 21);
```

При этом происходит **срезка** — копирование полей, находящихся только в родительском классе.

**NB:** класс-наследник не имеет доступа к **private**-членам родительского класса.

## 5.2 protected

Для доступа полей только наследникам используют модификатор доступа **protected**, но его использование потенциально нарушает инкапсуляцию, поэтому этот модификатор доступа разумно использовать для методов: таким образом можно выделить свой **protected**-интерфейс.

Можно также использовать такой хак (фичу ; ):

```
struct Parent {
    protected:
        int m;
};
struct Child : Parent {
    public:
        using parent::m;
};
```

Таким образом, можно изменить модификатор доступа (грязяжно).

На самом деле наследование происходит так:

```
struct Derived : <modifier> Base {...};
```

Если модификатор не указан явно, используется модификатор по умолчанию: для **struct** - **public**, для **class** - **private**.

Рассмотрим 3 случая:

- **public-наследование:** любой код, работающий с наследником, знает о том, что он унаследован от базового класса, а значит, он может вызывать все методы, которые унаследованы, а также приводить указатель или ссылку на производный класс на указатель или ссылку на базовый класс (публичные и защищённые данные наследуются, уровень доступа не меняется).
- **private-наследование:** информация о том, что наследник унаследован, доступна только внутри класса, а значит, только в пределах этого класса можно делать приведение типов и пользоваться унаследованными методами (все унаследованные данные становятся приватными).

- **protected-наследование:** информация о наследовании передаётся наследникам, и приведение типов и пользоваться унаследованными методами могут все наследники. (все унаследованные данные становятся защищёнными).

Как можно было делать перегрузки с функциями — так же можно делать перегрузки и с методами. При этом, если при вызове переменная не соответствует никакому типу в функциях, компилятор подберёт наиболее подходящую функцию.

```
struct Vector2d {
    Vector2d mult(double d) const {
        return Vector2d(x * d, y * d);
    }
    double mult(Vector2d const& p) const {
        return x * p.x + y * p.y;
    }
    ...
};
```

Перегрузка работает и при наследовании.

```
struct VectorShit : Vector2d {
    int mult(int i) const {
        return (int) x * (int) y * i;
    }
    using Vector2d::mult;
};
```

Если не добавить в наследника using, не будет перегрузки с методами родительского класса. Это происходит из-за name hiding: такой эффект можно получить, если в двух вложенных циклах объявить одну и ту же переменную - приоритет отдаётся последней.

**NB:** перегрузка происходит на этапе компиляции.

### 5.3 виртуальные методы

Что такое виртуальный метод — определение дать на ходу не очень. Потом может быть. Если совсем просто — это метод, который можно переопределить в классе-потомке.

Рассмотрим пример:

```
struct Person {
    virtual string name() const {
        return name_;
    };
    string action(){
        return "do nothing";
    }
    ...
};
struct Professor : Person {
    string name() const {
        return "Prof. " + name_;
    }
};
```

```
        string action(){
            return "teaching";
        }
};
...
Person xep("Donald");
Professor pr("Sorokin");
Person *p = &xep;
p->name(); // "Donald"
p->action(); // "do nothing"
p = &pr;
p->name(); // "Prof. Sorokin"
p->action(); // "do nothing"
```

Адрес конкретного метода, который будет вызван, будет зависеть от значения, хранящегося в указателе или ссылке.

**NB:** если метод был виртуальным в базовом классе, он будет виртуальным и в классе потомка.

**NB2:** виртуальный метод, вызванный в конструкторе или деструкторе, будет вести себя не виртуально (в целях защиты от обращения к неинициализированным полям).

**NB3:** если разместить определение виртуального метода за пределами класса, то ключевое слово `virtual` в определении указывать не надо.

**NB4:** конструктор виртуальным не бывает.

## 5.4 абстрактное или чистое виртуальное

Чистые виртуальные методы (абстрактные методы) — методы, у которых отсутствует реализация.

```
struct Person {
    virtual string name() const = 0;
    ...
};
```

Если класс содержит хотя бы один абстрактный метод, он становится абстрактным, и у него нельзя создать экземпляр. Зато можно создать классов-наследников, у которых будет реализация.

**NB:** Прямую попытку вызова отловит компилятор. Тем не менее, вызвать её можно, но тогда будет RTE (pure virtual function call)

```
#include <iostream>

class Base
{
public:
    Base() { init(); }
    ~Base() {}

    virtual void log() = 0;

private:
```

```
        void init() { log(); }
};

class Derived: public Base
{
public:
    Derived() {}
    ~Derived() {}

    virtual void log() { std::cout << "Derived created" << std::endl; }
};

int main(int argc, char* argv[])
{
    Derived d; // RTE (pure virtual function call)
    return 0;
}
```

**NB2:** Можно добавить такой функции тело и даже вызывать её:

```
class Base{
    int x;
public:
    Base(): x(1) {}
    virtual void print()=0;
};

void Base::print(){
    cout << x;
}

class Derived : public Base{
    int y;
public:
    Derived(): Base(), y(2) {}
    void print(){
        Base::print();
        cout << y;
    }
};

int main(){
    Base b; // CE, хоть тело и есть, бан в силе.
    Derived d;
    d.print(); // 12
    return 0;
}
```

Зачем нужна такая абстракция, если её как бы нет? Чтобы заставить наследников реализовать метод (это требование остаётся в силе). Плюс, всё ещё бан на создание объектов.



## 5.5 виртуальный деструктор

Виртуальный деструктор — обязательная вещь при использовании наследования. Если создадим указатель на класс-родителя, в который запишем класс наследника, в конце жизни вызовется только деструктор базового класса, если не использовать виртуальный деструктор. Будет утечка памяти, ОЖВП.

**NB:** Основное правило - если в классе есть хотя бы одна виртуальная функция - деструктор тоже должен быть виртуальным. При этом по умолчанию деструктор виртуальным не будет - его нужно объявить явно: `virtual Base() {...}`

**NB2:** При попытке объявить абстрактный деструктор компилятор будет ругаться на отсутствующее тело (справедливо, ведь наследники позовут деструктор родителя, а вызывать нечего). Однако, если такому деструктору написать тело - всё будет работать (хоть так делать и не принято). Профит - абстрактный базовый класс, при наследовании которого деструкторы будут виртуальными.

## 5.6 таблица виртуальных методов

Кстати, надо отметить, что перегрузка функций реализует(?) статический полиморфизм, а виртуальные методы — динамический полиморфизм (или полиморфизм времени выполнения).

Для реализации динамического полиморфизма используется таблица виртуальных методов, которая создаётся при инициализации. Таблица заводится для каждого полиморфного **класса** (класс, имеющий virtual-методы). Объекты полиморфного класса содержат указатель на таблицу виртуальных методов соответствующего класса.

```
|vptr|name_|age_|
    Person
|vptr|name_|age_|uni_|
    Student
```

Обычно указатель находится в начале, но обратиться к нему нельзя.

Вызов виртуального метода — вызов метода по адресу из таблицы. В коде создается номер метода в таблице для каждого класса. При этом `Person::vptr != Student::vptr`. Если виртуальный метод не реализован, вместо ссылки на функцию в таблице будет храниться адрес функции обработчика.

Для чистых виртуальных методов можно задать реализацию, но в таблице всё же будет храниться не эта реализация, но её можно будет вызвать, указав полный путь до неё (если это, конечно, нужно).

## 5.7 множественное наследование

Если сильно хочется, можно унаследоваться от нескольких классов.

```
struct Student : Person {...};
struct Worker : Person {...};
struct WorkingStudent : Student, Worker {...};
```

Но при этом в классе `WorkingStudent` будет храниться 2 экземпляра `Person`, и не гарантируется, что данные будут синхронизированы. При вызове метода, унаследованного от `Person` будет неоднозначность и компилятор не сможет понять, у какого из предков надо вызвать этот метод. СЕ.

При наследовании надо избегать наследования реализаций более, чем от одного класса и вместо этого, например, использовать интерфейсы (абстрактные классы без полей, у которых все методы абстрактные).

```
struct IWorker {...};  
struct Worker : Person, IWorker {...};  
struct WorkingStudent : Student, IWorker {...};
```

Но тогда надо будет переопределять методы 2 раза.

Успел не всё, плюс куда-то съехали отступы :(

## 6 Лекции 13/14

Эстафета передаётся ;)