

Конспекты по C++.

Семестр 2

Марк Тюков, Иван Кудрявцев

Весна 2020 года

Содержание

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

СОДЕРЖАНИЕ	2
4.2.3 Массив	7
4.2.4 Некорректное использование delete	7
5 Ссылки	8
5.1 Создание ссылки	8
5.2 Операции над ссылками:	8
6 Константы	9
6.1 Объявление	9
6.2 Что можно и нельзя	9
6.3 Константный указатель	9
6.4 Константная ссылка	9
7 Приведение типов	10
7.1 static_cast	10
7.2 Три запрещенных заклинания:	10
8 Введение в ООП	11
8.1 Что это такое?	11
8.2 Три волшебных слова в ООП	11
9 Инкапсуляция	11
9.1 Классы и структуры	11
9.2 Модификаторы доступа	11
9.3 Оператор “стрелочка”	11
9.4 Указатель this	12
9.5 Конструкторы	12
9.6 Деструкторы	13
9.7 Копирование, копирующее присваивание и правило трех	13
9.8 Константные методы	14
9.9 Списки инициализации в конструкторах	15
9.10 Friends	15
9.11 Explicit	15
9.12 Contextual conversial	15
9.13 Статические поля и методы	16
9.14 Pointers to members	16
10 Inheritance (наследование)	16
10.1 Модификатор PROTECTED	16
10.2 Размещение в памяти объектов наследников	17
10.3 Поиск имён при наследовании	17
10.4 Multiple inheritance	18
10.5 Приведение типов при наследовании	19
10.5.1 static_cast	19
10.5.2 reinterpret_cast	20
10.6 Виртуальные функции	20

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.¹

¹<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  
9     // Вместо этого так:  
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 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
```

10.5.1 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);
```

10.5.2 reinterpret_cast

Данный каст будет игнорировать приватность и все прочее.

10.6 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.