Конспекты по C++.

Семестр 2

Марк Тюков, Иван Кудрявцев

Весна 2020 года

Содержание

1 Введение в язык

1.1 Инструкции

1.1.1 Declarations (объявления)

Переменные

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

Примеры
[unsigned] int/long long/char/float/double/long double/bool
P.S.: size_t ≡ unsigned long long

Функции, структуры

```
void f(int x, double y){}
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 вычисляет выражение! 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 компилятор не может этого выполнить, например, потому что используемая функция просто отсутствует или типы не соответствуют (или еще что-нибудь)

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 получаем UB, в том числе можно получить RE (RE $\subset$ 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 объявлено, но не определено и используется

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

```
int x;

{
  int y; // выделение памяти
} // удаление из памяти ( на самом деле потеря адреса, что происходит с данными по тому адресу - неизвестно)

type* p;
*p; // унарная звездочка разыменовывания.

type* p = &x; // кладем в p адрес x
p + 1; p - 1;
```

3.1.2 Операции с указателями

1. Разыменование;
2. Сложение с числами;
3. Разность указателей.

void* - указатель на "сырую память то есть на данные неизвестного типа. 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.:

```
*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*)` ≠ `sizeof(a);`

3.3 Functions

Сигнатура - набор типов принимаемых аргументов.
Можно объявить несколько функций с разными сигнатурами

```
type f(int);
type f(double);
type f(int, int);
```

Эти функции могут возвращать данные разных типов
Компилятор при вызове таких функция 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 от double и от int.
P.S.: Читать в стандарте!!!

3.3.2 Указатель на функцию

```
int f(double);
int (*pf)(double) = f;

type f() {}
```

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

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

Потом нужно освободить память

```
delete p;
```

4.2.3 Массив

```
type* p = new type[n]; // запрашиваем память
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 UB (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:

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

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

P.S.: Java $\equiv \heartsuit$

```
type x;
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:

```
void swap(int& x, int& y) {
  int t = x;
  x = y;
  y = t;
}
```

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`:

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

4. Можно:

```
int x;
int& f() {
  return x; // x – глобальный
}
```

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)

```
int x = 1;
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 закончиться локальная видимость переменной.[1]

[1] https://habr.com/ru/post/186790/

7 Приведение типов

7.1 static_cast

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

Если преобразование не однозначно, то будет CE (неоднозначный каст).
Будет CE, 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:

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

Теоретически, это нужно, когда есть две функции:

```
f(int& x)
f(const int& x)
```

Если по какой-то причине захотим запустить f() от инт'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, в следующей посследовательности: const, static, static+const, reinterpret, reinterpret+const. Так что мы даже не узнаем, что сделалось

```
(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. Дальше 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();` ≡ `p->f();`

9.4 Указатель this

...

9.5 Конструкторы

Конструктор нужен для инициализации объектов некоторого класса.

```
C x = ... ;
// варианты инициализации
C x(...);
C x{...};
C x = C(...);
C x = {...}; // так еще можно делать в структуре без конструктора (если все поля паблик) – агрегатная инициализация.
```

Дальше в какой-то степени будем реализовывать класс строк

```
class String {
 public:

  String() { // конструктор
    str = new char[16]; // начальный размер
    size = 0;
  }

  String(size_t n) {
    ...
  }

 private:
  char* str;
  size_t size;
};
```

Конструктор по умолчанию – такое компилятор может сделать сам (он это делает, если мы этого не сделали), но он будет инициализировать по умолчанию все поля, что плохо в тех случаях, когда среди полей есть указатели или другие типы, для которых это плохо (рекурсия).
Первое правило – компилятор сам создает конструктор, если мы его не определили и не определили никакой другой конструктор. Но можно явно попросить его сгенерировать такой.

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

Конструкторы нужны в тех случаях, когда надо инициализировать не тривиально.
Можно поле создавать так (начиная с **C++11**):

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

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

```
String(...) : String(...) {
  ... // дополнительный код
}
```

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 можно сделать приватным, но тогда создать объект обычным способом нельзя будет.
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 конструктора 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

```
String s;
String s1;
s1 = s;
```

то здесь будет вызываться оператор присваивания.

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

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.

```
void f(...) const {
  ...
}
```

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:

```
mutable int counter;
```

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

9.9 Списки инициализации в конструкторах

```
struct S {
  int& x;
  const int y;

  S(int& x, int y) { // когда вошли в эту область видимости, поля должны быть уже проинициализированы, а ссылку невозможно проинициализировать так
  }

  // Вместо этого так:

  S(int& x, int y) : x(x), y(y) {} // здесь инициализация будет до входа в конструктор
};
```

Списки инициализации сохраняют нам одно копирование.

9.10 Friends

Иногда захочется, чтобы приватное поле было доступно.

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

```
  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.

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

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

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

9.12 Contextual conversial

Это конверсия в буль в ифах, форах, вайлах (под условиях). Такая конверсия игнорирует explicit (потому что не является неявных 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)
P.S.: 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 размер 3 полей: a, a, b (в данном случае 12, так как `int` = 4)

```
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 ≠ 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 CE
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() // CE, т.к. имя Granny inaccessible
```

```
1  ...
2
3  class Son : public Mom {
4  public void f(double) {
5    Granny g; // не сработает
6  }
7  public void f(double) {
8    ::Granny g; // сработает
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. Не надо так.

Почему? Из-за **проблемы ромбовидного наследования (diamond problem)**. Рассмотрим геометрические фигуры.
Имеем систему (стрелка ведет от сына к родителю)

Square → Rectangle; Square → Rombus; Rectangle → Parallelogram; Rhombus → Parallelogram;

Пусть в каждом лежит по инту. Тогда при создании квадрата создастся пять интов.
Синтаксис:

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

При множественном наследование (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

```
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

```
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()}

```
class Base {
  int x;
}

class Derived : public Base {
  int y;
}

Derived d;
Base* bp = &d;
Base& b = d;
b.f() // из Base
d.x++ // это изменит x и b, так как это разные имена одного и того же
Derived* dp = &b; // CE, потому что Derived потомок Base; чтобы так писать нужен явный каст
Derived& dd = b; // CE аналогично предыдущему


Base bb = d; // slicing
```

10.6.1 static_cast

```
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

```
static_cast<Granny&>(s);
```

10.6.2 reinterpret_cast

Данный каст будет игнорировать приватность и все прочее.
Множественное наследование?

```
Son s;
Father* fp = &s;
Father& f = s; // происходит правильный сдвиг указателей
Father& f = static_cast<Son&> // правильный сдвиг
Father f = reinterpret_cast<Son&> // НЕ правильный сдвиг
```

10.7 Виртуальные функции

Допустим, у нас есть координатная плоскость с двумя кругами: оба радиусом 1, в центре 1 и -1. Если мы опием 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:

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

Тогда если мы забудем что-то скопировать из сигнатуры, то будет CE и мы сможет подправить, чтобы программа нормально работа.

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()`.

```
Son s;
Mom& m = s;
// хотим сделать: Father& f = m; НО это CE!!! потому что тип мамы с типом папы не совместимы.
// тут на помощь приходит dinamic_cast
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 (CE). Правда, если классы вообще не совместимы, то будет CE. То есть RE в случаях сложной, запутанной иерархии. Если кастуем указатели, то вернется `nullptr`. В то же время repreint_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.

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

Хоть одна чисто виртуальная функция ⇒ 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

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

```
template<typename T>
class vector {
  ...
};
```

11.2 Typedef

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

11.3 Перегрузка шаблонной функции

Если есть разные версии шаблонной функции, то работает перегрузка.

```
template<typename U, typename V>
void f(U x, V y); // назовем это функция 1

template<typename U>
void f(U x); // функция 2

void f(int x, double y); // функция 3
```

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

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

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

Если у нас только

```
template<typename T>
void f();

f(); // CE
```

Можно делать шаблонный тип по умолчанию.

11.4 Специализация шаблонов

```
template<typename T>
class Class {

};


template<>
class Class<int> {

};

Class<char> c; // первого типа
Class<int> s // второго типа
```

11.4.1 Частичная специализация

```
template<typename T>
class Class<T*> {};
```

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

11.4.2 Порядок специализаций

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

```
template<typename T, typename U>
void f(T x, U y);

template<typename T>
void f(T x, T y);

template<>
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 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  typename<typename T>
7  struct is_same<T, T> { // специализация
8    static const bool value = true;
9  };
10
11 template<typenmae 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 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... Tale>
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

```
void print() {}

template<typename Head, typename... Tail>
void print(const Head& head, const& Tail& tail) {
  sizeof...(Tail)
  std::cout <<  head << " ";
  print(tail...);
}
```

12 Sequence containers

Vector, deque, list, forward_list

12.1 Vector

12.1.1 Объявление и прочие операции

```
vector<int> w;
vector<int> v(10); // в скобках первоначальный размер, в случае интов там будут лежать нули, то есть вызывается конструктор по умолчанию для каждого элемента
// если нет конструктора по умолчанию, то надо вторым параметром указывать значение
// для вектора векторов:
vector<vector<int>> v(100, vector<int>(100));

v.push_back(5); // добавление в конец
v.pop_back(); // удаление из конца; если из пустого, то UB!
v[1] = 3; // всё как с массивом
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

v.size(); // сколько элементов реально лежит элементов
v.resize(20); // изменение размера с дополнением элементов
v.capacity() // сколько памяти выделено
v.reserve() // выделение доп памяти

// pop_back() не уменьшает память!!
// поэтому в C++ 11 появился
v.shrink_to_fit(); // сжимает память до количества элементов в векторе

// !!!! sizeof(v) не зависит от количества элементов в векторе

v.back() // ссылка на последний элемент
v.front() // ссылка на первый элемент
// !!! если пустой, то UB

// более быстрый и эффективный метод, чем push_back()
v.emplace_back(); // см реализацию
```

12.1.2 Реализация

```
template<typename T>
class vector {
  private:
  T* arr;
  size_t sz;
  size_t cp;

  public:
  vector(size_t n, const T& valueT()): sz(n), cp(n) {
    arr = new char[n * sizeof(t)];
    for (size_t i = 0; i < n; ++i) {
      // placement new
      new(arr + i) T(value);
    }
  }

  void push_back(const T& value) {
    if (sz == cp) {
      // reallocate()
    }
    // placement new
    new(arr + sz) T(value)
  }

  void pop_back() {
    --sz;
    arr[sz].~T();
  }
};

template<typename... Args> // переменное количество аргументов
void emplace_back(const Args&... args) {
  ...
}
```

12.1.3 Вектор bool

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

```
template<>
class vector<bool> {
  // ...
  private:
    struct BoolProxy {
      int& b;

      BoolProxy operator =(bool b) {
        // bx = b; по сути, но мы работаем с интами
        // взять нужный бит и изменить его
        return *this;
      }
    };


```

```
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.

```
m.find(key); // возвращает итератор на элемент с данным ключом
// 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

14.1 Общий синтаксис

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

14.2 Как кидать исключение

```

int f() {
  throw std::out_of_range("текстовое описание"); // лежит в библиотеке stddexcept
  // throw – это оператор
  return 0;
}
```

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)

```
try {
// ...
} catch(const std::out_of_range& err) {
// ...
} catch(int x) {
// ...
} 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

```
struct MyStruct {
  int* p;

  MyStruct() {
    p = new int;
    f(p);
  }

  MyStruct(const MyStruct&) = delete;
  MyStruct& operator =(const MyStruct&) = delete;

  ~MyStruct() {
    delete p;
  }
};
```

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

```
struct VeryBad() {

  int x;
  VeryBad(int x): x(x) {}

  ~VeryBad() {
    f(x);
  }

};
```

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

```
int g() {
  VeryBad vb = 1;

  throw 1;
  return 0;
}

int main() {
  try {
    g();
  } catch (int x) {
    std::cout << x;
  }
}
```

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 крашается.
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

Когда нам нужно обернуть всё тело функции в `try`, то можно использовать более упрощенный синтаксис:

```
void f() try {

} catch() {

}
```

Это также имеет особые плюсы при работе с конструкторами:

```
struct S {
  int x;

  S(int x) try: x(x) {

  } catch(...) { // данный catch распространяется и на действие x(x)

  }
};
```

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.

```
void f(const std::map<int, int>& m) {
  // 1 метод — range based for / foreach

  // since C++11
  for (std::pair<int, int> p : m) { // тут еще можно убрать копирование заменив на ссылку и т.п.
    std::cout << p.first << " " << p.second << std::endl;
  }

  // 2 метод — константный итератор
  // в данном случае именно константый, потому что мап константный
  for (std::map<int, int>::const_iterator it = mc.begin(); it != m.cend(); ++it) {
    std::cout << it->first << " " << it->second << std::endl;
  }
}
```

15.1 Categories of iterators

Любой итератор позволяет разыминовывать себя.

15.1.1 Input

Это такие, которые позволяют один раз пройтись по себе, но не позволяют запись новое значение. Их можно сравнивать и инкрементировать.

15.1.2 Output

Это противоположность инпут-итераторов: их используют, чтобы записывать значения, но к их значению нельзя получить доступ. Они также умеют инкрементироваться, но их нельзя сравнивать

15.1.3 Forward

Умеют только вперед ++. Содржатся в `foward_list` и `unordered_map`

15.1.4 Bidirectional

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

15.1.5 Random access

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

15.1.6 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_catagory = //...
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 граммотно: 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 диапозон

```
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)

```
1  template<class Container>
2  struct back_insert_iterator {
3    Container& cont;
4    back_insert_iterator(Container& cont): cont(cont) {}
5
6    back_insert_iterator& operator++() { return *this; }
7
8    back_insert_iterator& operator*() { return *this; }
9
10   back_insert_iterator& operator=(const Container::value_type& x) {
11     cont.push_back(x);
12   }
13 };
```

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():

```
template<typename T>
void swap(T& a, T&b) {
  T t = std::move(a);
  a = std::move(b);
  b = std::move(t);
}
```

Таким образом, как нужно писать вектор:

```
vector& operator=(vector<T>&& v) {
  this->~vector();
  sz = v.sz;
  cp = v.cp;
  arr = v.arr;

  v.arr = nullptr;
}

vector(vector<T>& v): sz(v.sz), cp(v.cp), arr(v.arr) {
  v.arr = nullptr;
}
```

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

16.3 Move implementation

```
template<typename T>
std::remove_reference_t<T>&& move(T&& x) { // нельзя просто &&, об этом речь чуть позже
    return static_cast<std::remove_reference_t<T>&&>(x);
}
```

16.4 Rvalue references

Rvalue ссылки можно инициализировать только rvalue. Формально lvalue и rvalue определяются так: категории выражения (expression), для каждого выражения определятся, к какому типу он принадлежит.

Lvalue:

- identifier
- f() with return type lvalue reference
- cast to lvalue reference type
- =
- prefix ++
- unary *
- []
- n

Rvalue:

- literal (кроме строкового)
- f() with none reference return type OR RVALUE REFERENCE TYPE
- cast to non reference type or RVALUE REFERENCE TYPE
- +
- -
- binary *
- /
- %
- «, »
- &, |, ;
-
- &&, ||

Важно понимать, что следующий код неправильный:

```
1  int&& r = 0; // так можно
2  int&& rr = r; // тут CE, потому что тип правой части rvalue reference, но является lvalue
```

16.5 Perfect forward problem

Если есть некоторая функция, принимающая как и rvalue так и lvalue, то хорошо бы и дальше передавать ту же категорию, что и получили. Поэтому введен universal reference (вроде бы, это неофициальное название) — такая вещь, которая может биндиться и к lvaue и rvalue. Такой считается ссылка, которая имеет вид “шаблонный параметр функции + &&”. Чтобы пробрасывать дальше, сохраняя их категорию, существует метод `std:forward`

16.6 Reference collapsing

```
template<typename T>
void f(T&& x) {
  // reference collapsing:
  // type& + & -> type&
  // type&& + && -> type&&
  // type& + && -> type&
  // type&& + & -> type&
}

int main() {
  f(0); // вызываем от rvalue => T = int, decltype(p) = int&&
  int x = 0;
  f(x); // вызываем от lvalue => T = int&, decltype(p) = int&
}
```

Поэтому в муве мы и используем `remove_reference_t`: мув принимает универсальную ссылку, а должна возвращать rvalue.

16.7 method `forward`

16.7.1 Использование

```
template<typename... Args>
void emplace_back(Args&&... args) {
  // ...
  new(arr + sz) T(std::forward<Args>(args)...);
}
```

16.7.2 Реализация

```
template<typename T>
T&& forward(std::remove_reference_t<T>& x) {
  return static_cast<T&&>(x);
}
```

Стоит подробнее изучить, как это работает в различных случаях, чтобы лучше разобраться.

16.8 glvalue, lvalue, xvalue, rvalue, prvalue

Начиная с **C++11** существует на самом деле 5 категорий (всё выше остается таким же)
glvalue — это либо lvalue либо xvalue (*generalised lvalue*)
rvalue — это либо xvalue, либо prvalue
prvalue — это чистый rvalue (*pure rvalue*)
xvalue — это rvalue с некоторыми схожестями с lvalue (*expiring value*)
В перечислении выше (), xvalue выделены капсом.
xvalue от prvalue отличается тем, что это полиморфный тип.

16.9 RVO - return value oprtimisation

```
T f() {
  return T(); // временный объект сразу (без копирования или мува) помещается в возвращаемый объект
}
```

Это называется **copy elision**

16.10 Ref-qualifiers

С **C++11** мы можем написать такую функцию в классе, что она вызывается от левого аргумента lvalue:

```
struct S {
  int h() & { // только для lvalue
    return 0;
  }

  int h() && { // только для rvalue
    return 1;
  }
};
```

P.S.: Если нет ни одного амперсанда, то подходит для любых value.

16.10.1 Практическое применение ref-qualifiers

Допустим, у нас есть большие объемы данных (в полях) и есть метод `getData()` – получить данные данные. И вот тут проблема: данные большие, значит мы их скопируем (долго и тд). Но мы могли бы использовать мув-семантику и просто переместить их, если мы знаем, что объект скоро "подохнет".

17 Умные указатели

17.1 Мотивация или постановка проблемы

Рассмотрим следующую проблему: есть динамически выделенная память (или какой-нибудь захваченный ресурс), а потом вызывается какая-нибудь функция, которая кидает исключение, и тогда мы можем "не успеть"освободить память. Или мы просто забыли забыли прописать `delete`, who knows

17.2 Решение в C++03

```
template<typename T>
struct auto_prt {
  T* ptr;
  auto_ptr(T* ptr): ptr(ptr) {}
  ~auto_prt() { delete ptr; }
};
```

Такая простая структура уже решает вышеупомянутую проблему, но в **C++11** появились решения лучше, а с **C++17** , кажется, вообще удалена. У данной структуры свои проблемы — у неё нарушено правило трёх, например.

17.3 unique_prt

```
template<typename T>
class unique_ptr {
  private:
  T* ptr;
  public:
  unique_ptr(T* ptr): ptr(ptr) {}

  unique_ptr(const unique_ptr<T> &p) = delete;
  unique_ptr<T>& operator=(const unique_ptr<T> &p) = delete;

  unique_ptr(unique_ptr<T> &&p): ptr(p.ptr) {
    p.ptr = nullptr;
  }

  unique_ptr<T>& operator=(unique_ptr<T> &&p) {
    delete ptr;
    ptr = p.ptr;
    p.ptr = nullptr;
  }

  T& operator*() {
    return *ptr;
  }

  T* operator->() {
    return ptr;
  }

  ~unique_ptr() { delete ptr; }
};
```

Таким образом, данный указатель можно хранить в контейнерах (ура)
P.S.: в реальности `unique_ptr` принимает два шаблонных параметра: второй — это `typename Deleter`, у которого предопределен оператор `()` и он вызывается (как функция) в деструкторе (а в свою очередь дефолтный `Deleter` вызывает `delete`)

17.3.1 Почему unique?

Ответ прост: как видно из реализации, одновременно на одно и то же может указывать только один объект, и поэтому можно не бояться удалять его в деструкторе.

17.3.2 make_unique

(При первом прочтении сначала стоит изучить следующий пункт)
Данная функция — аналог `make_shared` только для `unique_ptr`.

```
template<typename T, typename... Args>
unique_ptr<T> make_unique(Args&&... args) {
```

```
3     return unique_ptr<T>(new T(std::forward<Args>(args)...));
4   }
5
```

Но, как может показаться на первый взгляд, она не решает аналогичной проблемы как в случае с `shared_ptr`. Рассмотрим пример:

```
1   struct S {
2     int x;
3     double y;
4     S(int x, double y): x(x), y(y) {}
5   };
6
7   void foo(const unique_ptr<S>& p, int x) {}
8
9   int getInteger() { return 0; }
10
11  // хотим передать в foo() unique_ptr, а вместо числа будет передаваться значение какой-то другой функции
      getInteger()
12  foo(unique_ptr<S>(new S(1, 2.0)), 0);
```

Вспомним, для чего нужен `unique_ptr`: чтобы точно освободить выделенную память в деструкторе. Так что же тут может пойти не так? Разгадка в том, что порядок выполнения вызова аргументов не определен стандартом

17.4 shared_ptr

Что если мы всё-таки хотим уметь копировать (и чтобы он удалялся только тогда, когда удаляем последнюю копию)?

```
1   template<typename T>
2   class shared_ptr {
3     private:
4     T* ptr;
5     int* counter;
6     public:
7     shared_ptr(T* ptr): ptr(ptr), counter(new int(1)) {}
8
9     shared_ptr(const shared_ptr<T> &p): ptr(p.ptr), counter(p.counter) {
10      ++*counter;
11    }
12
13    public:
14
15    shared_ptr<T>& operator=(const shared_ptr<T> &p) = delete;
16
17    shared_ptr(shared_ptr<T> &&p): ptr(p.ptr), counter(p.counter) {
18      p.ptr = nullptr;
19      p.counter = nullptr;
20    }
21
22    shared_ptr<T>& operator=(shared_ptr<T> &&p) {
23      delete ptr;
24      ptr = p.ptr;
25      p.ptr = nullptr;
```

```
    }

    T& operator*() {
      return *ptr;
    }

    T* operator->() {
      return ptr;
    }

    ~shared_ptr() {
      if (*counter > 1  ) {
        --*counter;
      } else {
        delete ptr;
        delete counter;
      }
    }
  };
```

17.4.1 std::make_shared

Запрос динамической памяти в конструкторе `shared_ptr` — очень дорогая (по времени) операция.

```
  struct S {
    int x;
    double y;
    S(int x, double y): x(x), y(y) {}
  };

  void g() {
    shared_ptr<S> p = make_shared(1, 2.0);
  }
```

```
template<typename T, typename... Args>
shared_ptr<T> make_shared(Args&&... args) {
  char* ptr = new char[sizeof(T) + sizeof(int)];
  new(ptr + sizeof(int)) T(std::forward<Args>(args)...); // вызывается приватный конструктор (см ниже)
  return shared_ptr<T>(ptr + sizeof(int), ptr);
}
```

Данная функция должна быть дружественной для `shared_ptr`.

```
  // тот самый приватный конструктор

  private:
    shared_ptr(T* ptr, int* counter) : ptr(ptr), counter(counter) {}
```

И поэтому, нам на самом деле надо хранить то, как элемент был создан (через `make_shared` или нет), что можно делать, например, рядом со счетчиком (достаточно первого бита счетчика для этого). И соответственно, нужно расширить деструктор на случай, когда создали через `make_shared`:

```
~shared_ptr() {
  if (*counter > 1  ) {
    --*counter;
  } else if (...) { // тут необходимая проверка
    delete ptr;
    delete counter;
  } else {
    ptr->~T();
    delete[] reinterpret_cast<char*>(counter);
  }
}
```

17.4.2 Более правильная реализация

(При первом прочтении к данной части стоит вернуться после следующего пункта)
На самом деле в `shared_ptr` счетчик храниться указателем не на инт, а на следующую структуру:

```
struct Counter {
  int shared_count = 0;
  int weak_count = 0;
};
```

Что несет соответствующие изменения в методы, в частности в деструктор:

```
~shared_ptr() {
  if (*counter > 1  ) {
    --*counter;
  } else if (...) { // тут необходимая проверка
    delete ptr;
    if (counter->weak_count == 0 ) delete counter;
  } else {
    ptr->~T();
    if (counter->weak_count == 0) delete[] reinterpret_cast<char*>(counter);
    // counter надо удалять в деструкторе weak_counter
  }
}
```

17.5 weak_ptr

Данный указатель нужен, чтобы решить проблему циклических ссылок.
Его нельзя разыменовать (под ним может быть всё удалено). Но можно попросить сделать `shared_ptr` на то же самое.

17.5.1 Реализация

```
template<typename T>
class weak_ptr {

  Counter* counter; // та самая структура из shared_ptr

```

```
6     public:
7
8       weak_ptr() const shared_ptr& p);
9
10      bool expired() const; // проверяет, "жил"ли данный объект
11      shared_ptr<T> lock() const; // если жив, то можно создать shared_ptr, иначе получим nullptr
    (возможно, исключение, надо проверить)
12
13      size_t use_count() const; // сколько shared_ptr указывают на наш объект (есть и у shared_ptr)
14
15  };
```

18 Compile-time checks, SFINAE

18.1 Простой пример

SFINAE — Substitution Failure Is Not An Error
Представим, что есть две функции (с одинаковым названием).

```
1
2   struct S {
3     using type = int;
4   };
5
6   void f(...) {
7     std::cout << 0
8   }
9
10  template<typename T>
11  typename T::type f(T a) {
12    return 0;
13  }
14
15  f(S());
16  f(0);
```