

# **Язык программирования C++**

## **Конспект лекций**

# Содержание

Лекция 1.	3
Лекция 2.	4
Лекция 3.	5
Лекция 4.	6
Этот конспект не завешен.	6
Функции.	6
Лекция 5.	7
Лекция 6.	8
Строки.	8
Структуры.	10
Инициализация структуры.	10
Обращаться к полям структуры.	11
Лекция 7.	12
Классы	12
Модификаторы доступа.	12
Инициализация класса	13
Деструкторы	14
Лекция 8.	15
Динамическая память	15
Лекция 9.	16
Лекция 10.	17

## **Лекция 1.**

TODO

## **Лекция 2.**

TODO

## **Лекция 3.**

TODO

## Лекция 4.

Этот конспект не завешен.

### Функции.

Программы на C++ выглядят, как множество функций, которые друг друга вызывают, поэтому фактически невозможно запустить код вне функций.

Уже знакомая нам функция — `main`. Программа запускает ее автоматически, а в конце получает от нее: успешно ли выполнялась программа.

Очевидно, создавать можно и свои функции. Рассмотрим пример:

```
int add(int a, int b) {  
    return a + b;  
}
```

Сначала задается тип возвращаемого результата функции, после – название, а наконец аргументы, каждый с своим типом данных. Каждая функция заканчивается ключевым словом `return`, который возвращает значение функции.

Команда `return` не обязательно должна стоять в самом конце кода функции. Например, их можно поставить несколько, некоторые в середину

```
int add(int a, int b) {  
    if (a == 0 && b == 0){  
        return 0;  
    }  
    return a + b;  
}
```

Важно помнить, что на `return` функция возвращает свое значение и заканчивает выполнение.

Есть функции, которые ничего не возвращают:

```
void foo() {  
    std::cout << "nothing" << std::endl;  
}
```

Вызываются функции любые функции одинаково:

```
int main() {  
    std::cout << add(1, 2) << std::endl;  
    std::cout << add(3, 3) << std::endl;  
  
    foo();  
    foo();  
}
```

На самом деле, функция `main` тоже принимает аргументы...

TODO

## **Лекция 5.**

TODO

## Лекция 6.

### Строки.

Немного общее про хранение символов. Поскольку символов много, а их все надо как-то однозначно определить, придумали кодировки. Они позволяют хранить в памяти просто номер код символа, а программа уже сама будет по этому коду определять, какой символ рисовать.

Например, есть кодировка `utf-8`. Она хранит символы Unicode и занимает 8 бит или 1 байт. Коды символов имеют полную обратную совместимость с 7-битной кодировкой ASCII.

В C++ по умолчанию используется `utf-8`.

В C++ строку можно хранить, например, как просто массив элементов типа `char`. Будет не более 256 различных значений.

```
char str[] = {'H', 'e', 'l', 'l', 'o'};
std::cout << str << std::endl;
```

На этот код санитайзер будет ругаться. Для вывода этого массива в `std::cout` передается указатель на первый символ, `char* ptr = &str[0]`. Выходит так, чтобы вывести строку, программа должна пройти по памяти, начиная с адреса начала строки, однако она не знает, где остановиться, и выходит за его пределы. Но далее программа может просто дойти до конца выделенной ей памяти и сломаться — что она и делает. В этот момент санитайзер предупреждает, что мы вышли за пределы памяти.

Тогда можно выводить строку такой функцией:

```
void print(const char* str, size_t size) {
    while (size > 0) {
        std::cout << *str << std::flush;
        ++str;
        --size;
    }
}
```

Или можно задавать размер заранее, прямо в строке:

```
// const char str[] = {0, 0, 0, 5, 'H', 'e', 'l', 'l', 'o'}; //первые 4 байта -- размер строки

void print(const char *str) {
    size_t size = 0;
    for (int i = 0; i < 4; ++i) {
        size <= std::numeric_limits<unsigned char>::digits;
        size |= *str++;
    }

    while (size > 0) {
        std::cout << *str << std::flush;
        str++;
        size--;
    }
}
```



Теперь о строках. В каждой строке, помимо самих символов строки, есть и специальный символ конца строки: `'\0'`. Поэтому, определив строку через двойные кавычки, мы можем вывести так:

```
void print0(const char* str) {
    while (*str != '\0') {
        std::cout << *str++;
    }
}

int main() {
    char str[] = "abcde";
    print0(str);

    return 0;
}
```

Однако, и у строк есть свои минусы: мы не можем знать длину строки за  $O(1)$ . Чтобы узнать длину, мы должны пройти по всей строке, от адреса начала, до символа `'\0'`.

## Структуры.

Хорошо бы уметь хранить строку и ее размер в одном месте, в одной структуре. Для этого есть структуры.

```
struct StringRef {
    const char* begin;
    size_t size;
}

struct OwningString {
    char str[100]; // здесь 100 -- длина строки, следовательно займется память на 100 чаров
                  // и тут еще будет char padding[7]. (выравнивание)
    size_t size;
}
```

Тогда остальные функции мы можем переписать так:

```
void print(StringRef str) {
    for (size_t i = 0; i < str.size; ++i) {
        std::cout << str.begin[i] << std::endl;
    }
}

int main() {
    const char* ptr = "abcdefg"; // где-то в памяти лежит это строка.
    StringRef str{ptr, 7};
    OwningString str2{"ABCD", 4};
}
```

## Инициализация структуры.

Можно просто сначала инициализировать и потом поименно прописывать каждое поле:

```
int main() {
    StringRef str;
    str.size = 3;
    begin = "123"
}
```

Можно сразу прописывать при инициализации, но тут важно соблюдать порядок, как они расположены в определении, и заполнять все значения:

```
int main() {
    StringRef str{"123", 3};
    // StringRef str = {"123", 3}; // Можно еще так
}
```

Начиная с C++20 появилась новая схема:

```
int main () {
    StringRef str{
        .begin = "123",
        .size = 3,
    } // Тут надо писать их в том порядке, в каком они задекларированы в определении структуры
}
```

## Обращаться к полям структуры.

Самое простое – через точку:

```
std::cout << str.size << std::endl;
```

Если у нас есть указатель на структуры:

```
StringRef ref;  
StringRef* ptr = &ref;  
  
(*ptr).size = 42;  
// или  
ptr->size = 13;  
// или  
const auto& [begin, size] = ref; // называется распаковка.  
// Вся структура разбивается на переменные.  
// В квадратных скобках пишутся имена, по которым можно найти переменную.
```

## Лекция 7.

### Классы

```
struct Point {  
    int x;  
    int y;  
};
```

Пусть есть структура Rect :

```
struct BadRect {  
    int width = 0;  
    int height = 0;  
    int area = 0;  
};  
  
int GetRectAreaSlow(const BadRect& rect) {  
    // ...  
}
```

Структуры – это всего лишь набор переменных, которые никак не связаны и независимы, но если нам потребуется какая-то связь, то надо использовать классы.

```
class Rect {  
public:  
    int SetWidth(int new_width) {  
        width_ = new_width;  
        UpdateArea();  
    }  
  
    int Area() const { // const не позволяет менять приватные переменные  
        return area_;  
    }  
  
private:  
    void UpdateArea() {  
        area_ = width_ * height_;  
    }  
  
private:  
    int width_ = 0;  
    int height_ = 0;  
    int area_ = 0;  
};
```

Классы практически не отличаются от Структур. Единственное отличие – люди договорились, что структура считается набором полей, а классы еще имеют связи между полями.

### Модификаторы доступа.

- **private** : поля и функции можно менять/читать/вызывать только изнутри класса. Другими словами, к ним нет доступа снаружи.
- **public** : можно менять извне; методы в этом разделе можно вызывать снаружи.
- **protected** : ...

Константные поля;

```
public:
```

В константной функции нельзя менять приватные переменные.

```
int foo() const {  
  
}
```

Если в разделе `private` объявить переменную как `mutable int area`, то ее можно менять даже в `const` функции.

```
// код про кэш; пример для использования mutable переменных.
```

Таким образом мы можем обращаться к `RectId` как к типу данных, равному по свойствам `uint64_t`

```
class RectDataBase {  
public:  
    using RectId = uint64_t;  
  
    Rect GetRect(RectId) {  
        // ...  
    }  
}
```

## Инициализация класса

Конструктор вызывается автоматически, сразу при создании объекта. Код просто не скомпилируется, пока в конструктор не передадут все важные переменные.

```
class Rect {  
public:  
    Rect(int width, int height) { // конструктор  
        width_ = width;  
        height_ = height;  
        UpdateArea();  
    }  
  
private:  
    void UpdateArea() {  
        area_ = width_ * height_;  
    }  
  
private:  
    int width_ = 0;  
    int height_ = 0;  
    int area_ = 0;  
}
```

```
Rect(int width, int height) // другой вид конструктора  
    : width_{width}  
    , height_{height}  
    {}
```

Можно пользоваться также и перегрузкой функций и тогда компилятор сам будет выбирать нужный конструктор.

Если есть функция, принимает в себя объект класса, то вызов функции может выглядеть неочевидно:

```
void foo(Rect r){  
    ...  
}  
  
int main() {  
    foo(123);  
}
```

Тогда в конструкторе можно указать ключевое слово `explicit`, и тогда мы будем обязаны в функции указать иначе

```
int main() {  
    foo(Rect{123})  
}
```

## Деструкторы

Как понятно из названия, вызывается в конце существования объекта.

```
class Noisy {  
public:  
    Noisy(int idx) {  
        // что выполнится при создании объекта  
    }  
    ~Noisy() {  
        // что выполнится в конце существования объекта  
    }  
}
```

Объект удаляется, как и переменные, в конце блока с `{...}`.

## Лекция 8.

### Динамическая память

1. Глобальная память — снаружи всех функций, доступ к ним есть отовсюду. Разрушаются переменные только в конце выполнения программы. Чаще всего надо избегать использования глобальных переменных.
2. Автоматический вид памяти. Компилятор сам — автоматически — разрушает переменные; также автоматически выделяет память.
3. Динамическая память — объекты живут в памяти ровно столько, сколько прописано в программе.

```
for (int i = 0; i < 1000; ++i) {  
    int* ptr = new int{123};  
    delete ptr; // удаляем то, на что указывает указатель  
}  
  
// объект, который живет в автоматической памяти;  
// указатель живет в автоматической памяти;  
// сам объект указателя живет после delete;
```

Ключевое слово (оператор) `new` создает объект и выделяет для него автоматическую память. Когда мы создали объект и потеряли к нему указатель, то мы не больше доступа к нему и не можем удалить. Называется утечкой памяти.

Вернуть из блока `{...}` тот указатель:

```
int* allocate() {  
    int* ptr = new int{123};  
    return ptr  
}  
  
int main() {  
    int* ptr = allocate();  
    delete ptr;  
  
    return 0;  
}
```

Динамическая память живет не на стеке, а на “куче”, поэтому там можно хранить большие данные.

```
class UniquePtr {  
    UniquePtr(int* ptr) {  
  
    }  
}
```

```
std::unique_ptr<int> ptr_main = AllocateSmart();  
std::unique_ptr<int> ptr_main2 = ptr_main; // так нельзя  
  
std::shared_ptr<int> s1{new int{42}};  
std::shared_ptr<int> s2;  
std::shared_ptr<int> s3;
```

## **Лекция 9.**

TODO



## Лекция 10.

TODO