

# C++

*Fall 2020*

*compscicenter.ru*

*Башарин Егор*

*eaniconer@gmail.com*  
*<https://t.me/egorbasharin>*

# Лекция III

Statements. References. Heap. Linkage.

# Основные конструкции

# Statements

*(Утверждения)*

---

Части программы, которые выполняются последовательно.

Пример:

```
int main() {  
    int i = 0;           // statement 1  
    std::cin >> i;       // statement 2  
    return i;            // statement 3  
}
```

# Types of statements

- expression statements
- compound statements
- selection statements
- iteration statements
- jump statements
- declaration statements

# Expression statements

Syntax:

```
[expression] ;
```

- Выражение — комбинация операторов и операндов ([Click me](#))
- Null statement — в случае отсутствия выражения.  
(Квадратные скобки в синтаксисе указывают на опциональность)

Пример:

```
n += 3;           // expression statement
std::cout << n;   // expression statement
;                 // null statement
```

# Compound statements

Syntax:

```
{ [statements...] }
```

Последовательность утверждений, обернутых в фигурные скобки.

Пример:

```
int main() {  
    {                // start of compound statement  
        int n = 10;  // ???  
        std::cout << n; // expression statement  
    }                // end of compound statement  
}
```

# Selection statements

Syntax (since c++17):

```
if [constexpr] ([init-statement] condition) statement
```

```
if [constexpr] ([init-statement] condition) statement else statement
```

```
switch ([init-statement] condition) statement
```

- `init-statement`:
  - expression statement
  - simple declaration ([Click me](#))
- `constexpr` пока не рассматриваем



# if-else

```
if ([init-statement] condition) statement else statement
```

- statement — любое утверждение
- condition
  - expression statement, результат которого может быть приведен к типу `bool` (`contextually converted to bool`)
  - объявление переменной `non-array` типа с `brace-or-equals` инициализацией

# switch

```
switch ([init-statement] condition) statement
```

- condition
  - выражение целочисленного типа [или типа перечисления]\*
  - [выражение типа, которое контекстуально неявно преобразуется к целочисленному типу или типу перечисления]\*
  - объявление переменной non-aggrau типа с brace-or-equals инициализацией (см ограничения на тип в первых двух пунктах)

# Labels

(необходимо для дальнейшего понимания `switch`)

Любое утверждение (`statement`) можно пометить именованной меткой

Syntax:

```
identifier: statement           (1)
```

```
case const_expression: statement (2)
```

```
default: statement             (3)
```

(1) используется с `goto`

(2), (3) специальные метки, которые используются с утверждениями внутри тела `switch`

# switch

```
switch ([init-statement] condition) statement
```

statement — любое утверждение (обычно compound)

Внутри statement может использоваться:

- Метка `case const_expression`:
  - любое количество меток без дубликатов
  - значение `const_expression` известно на этапе компиляции, а тип совпадает с типом из `condition`
- Метка `default`: (не более одной)
- `break`; для выхода из тела statement

# switch

## ОПИСАНИЕ

Вычисляется значение `condition`

Если такое значение есть среди `const_expression` у `case` меток, то управление передается к утверждению после соответствующей метки, иначе управление передается к метке `default`.

Утверждение `break`; приводит к выходу из тела `switch`

*Вопрос: выполнится ли код в теле, если там не будет меток?*

# Iteration statements

`while (condition) statement` (1)

`do statement while (expression)` (2)

`for (init-statement [condition]; [expression]) statement` (3)

`for ( for-range-decl : for-range-init ) statement` (4)

Пример для (4):

```
// принцип работы range-based for loop разберем в следую
int arr[] = {1, 2, 3, 4};
for (int item : arr) {
    std::cout << item << " ";
}
```

# Jump statements

`break;` - выход из тела цикла или тела `switch`

`continue;` - переход к следующей итерации цикла

`return [expression];` - прерывает текущую функцию и возвращает значение

`goto identifier;` - передает управление по метке

# Declaration statements

Необходимы, чтобы ввести новые символы (идентификаторы) в программе.

Пример:

```
int n = 1;  
double a = 1, b = 2;
```



# Ссылки

```
int object = 32;  
int& refToObject = object;  
const int& constRefToObject = object;
```

Задаёт псевдоним к уже существующему объекту

Инициализация обязательна

Можно представлять себе, как постоянно разыменованный указатель

# Инициализация константных ссылок

Неконстантная ссылка может быть проинициализированна только объектом, который не считается временным.

Константная ссылка продлевает время жизни временных объектов:

- результат сохраняется во временный объект
- время жизни объекта ограничено временем жизни ссылки
- применимо только локальных константных ссылок

```
const int& r = 1 + 2;
```

# Dangling references

Ссылки на объекты, которые уже были уничтожены.

Доступ к объектам по таким ссылкам — UB.

# Динамическая память

# Мотивация

- Ограниченность стека

```
double m [10*1024*1024] = {}; // 160 Мб  
// Скорее всего программа упадет с ненулевым кодом возвр
```

- Время жизни локальных объектов ограничено телом функции
- При создании массива, его размер не всегда известен на этапе компиляции, а использование VLA не является стандартом C++.

# О динамической памяти

- Выделять и освобождать память необходимо вручную
- Память выделяется в куче (не путать с одноименной структурой данных)

# Выделение и освобождение памяти

- `new/delete` — для одиночных значений
- `new[]/delete[]` — для массивов

```
int* objectInHeap = new int(10);    // Выделение памяти
delete objectInHeap;                // Освобождение памяти

int* arrayInHeap = new int[5]();    // Выделение памяти
delete [] arrayInHeap;              // Освобождение памяти
```

# Распространенные проблемы

- Утечка памяти (Memory Leak)
- Повторное освобождение памяти (Double free)
- Использование непарного выражения освобождения.  
Например память выделена с помощью `new[]`, а освобождена с помощью `delete`.



# Динамические массивы

```
int* arr1 = new int[10]; // значения элементов могут быть  
delete [] arr1;  
  
int* arr2 = new int[10](); // Массив будет инициализирован  
delete [] arr2;
```

# Двумерный динамический массив

```
// zero matrix 10x5
const size_t N = 10;
const size_t M = 5;
int** matrix = new int*[N]; // массив из N указателей на
for (size_t i = 0; i < N; ++i) {
    // Создание динамического массива размера M и заполн
    int* arr = new int[M]();

    // matrix[i] -- это указатель на int
    matrix[i] = arr;
    // теперь matrix[i] указывает на первый элемент масс
}

/// освобождение памяти остается для самостоятельной раб
```

# Функции

```
#include <cmath>
#include <iostream>

double someFormula(int i, double x) {
    double result = std::asin(1 / std::sqrt(3)) * x;
    if (i > 500) { result += 3.4 }
    return result;
}

int main() {
    int i = 501;
    double d = 44.4;
    std::cout << someFormula(i, d);
    return 0;
}
```

# Вызов функции

В зависимости от соглашения о вызове определяется:

способ передачи аргументов: регистры и/или стек

порядок размещения аргументов в регистрах/стеке

ответственный за очистку стека: callee/caller

способ передачи результата в точку вызова

способы возврата (передачи управления) в точку вызова

# static

```
double someFormula(int i, double x) {  
    static const double coef = std::asin(1 / std::sqrt(3)  
    double result = coef * x;  
    if (i > 500) { result += 3.4 }  
    return result;  
}
```

# pass by value, by reference, by pointer

```
#include <cassert>

void func(int value, int& ref, int* ptr) {
    value = 10;
    ref = 20;
    if (ptr) { *ptr = 10; }
}

int main() {
    int a = 1, b = 2, c = 3;
    func(a, b, &c);
    assert(a == 1);
    assert(b == 20);
    assert(c == 10);

    return 0;
}
```

# main

точка входа в программу

```
int main(int argc, char* argv[]) {  
    return 0; // не обязателен, 0 by default.  
}
```

argc - положительное число;  
число аргументов, переданных программе при запуске;  
argv - массив `argc + 1` указателей на строки, представляющих аргументы;  
последний указатель нулевой;  
строки изменяемые

# Глобальные переменные

```
// main.cpp
#include <iostream>

const int GLOBAL_CONST = 1;
int globalVar = 2;

int getN() { return globalVar + GLOBAL_CONST; }

int main() {
    std::cout << getN() << std::endl;;
    globalVar = 3;
    std::cout << getN() << std::endl;
    return 0;
}
```



# Глобальные переменные

- по умолчанию инициализируются нулем
- инициализируются в порядке появления в единице трансляции
- между разными единицами трансляции порядок не определен

# Области видимости

```
int i = 10;
int main() {
    int i = 11;
    std::cout << i << " ";
    std::cout << ::i << " "; // access to global

    {
        int i = 12;
        std::cout << i << " ";
        std::cout << ::i << " "; // access to global
    }

    std::cout << i << " ";
}
// Output: 11 10 12 10 11
```

# header usage

```
// vars.hpp
#pragma once
const int GLOBAL_CONST = 1;
int globalVar = 2;
```

```
// main.cpp
#include "vars.hpp"
int main() { }
```

```
// a.cpp
#include "vars.hpp"
```

```
$ clang++ main.cpp a.cpp
duplicate symbol '_globalVar' in:
  /var/folders/qw/h2f_rj2x5m54pqqwrzn1kt94zk09sb/T/main-9815ac.o
  /var/folders/qw/h2f_rj2x5m54pqqwrzn1kt94zk09sb/T/a-475cb4.o
ld: 1 duplicate symbol for architecture x86_64
clang: error: linker command failed with exit code 1
```

# FIX: Function, static

```
// vars.hpp
#pragma once
const int GLOBAL_CONST = 1;

int& globalVar(); // declaration
```

```
// vars.cpp
#include "vars.hpp"
int& globalVar() { // definition
    static int var = 2;
    return var;
}
```

```
// main.cpp
#include "vars.hpp"
int main() { }
```

```
// a.cpp
#include "vars.hpp"
```

# FIX: inline (C++17)

```
// vars.hpp
#pragma once
const int GLOBAL_CONST = 1;
inline int globalVar = 2;
```

```
// main.cpp
#include "vars.hpp"
int main() { }
```

```
// a.cpp
#include "vars.hpp"
```

```
clang++ -std=c++17 main.cpp a.cpp
```

# FIX: Extern

```
// vars.hpp
#pragma once
const int GLOBAL_CONST = 1;
extern int globalVar; // declaration
```

```
// vars.cpp
#include "vars.hpp"
int globalVar = 2; // definition
```

```
// main.cpp
#include "vars.hpp"
int main() { }
```

```
// a.cpp
#include "vars.hpp"
```

# Declaration / definition

```
// a.cpp
```

```
// declarations:
```

```
void foo(int);  
extern int var;
```

```
// definitions (and declarations as well):
```

```
void foo(int) { /* code */ }  
int var = 3;
```

# linkage

```
// a.cpp

// external linkage (символы видны вне единицы трансляции)
void f() { }
int i;
extern const int j = 300; // force extern for const

// internal linkage (символы видны внутри единицы трансляции)
static void g() { }
const int ci = 1; // by default
static int si = 2;
```

```
$ clang++ -c a.cpp
$ nm a.o
0000000000000000 T __Z1fv
0000000000000070 S _i
0000000000000008 S _j
```