C++

Башарин Егор

eaniconer@gmail.com

Лекция III

References.

Heap.

Statements.

Functions.

Ссылки

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

Задает 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 Mb
// Скорее всего программа упадет с ненулевым кодом возврата
```

- Время жизни локальных объектов ограничено телом функции
- При созданнии 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)
- Use after 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 любыми
delete [] arr1;

int* arr2 = new int[10](); // Массив будет инициализирован нулями
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 int
for (size_t i = 0; i < N; ++i) {
    // Создание динамического массива размера M и заполненного нулями
    int* arr = new int[M]();

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

/// освобождение памяти остается для самостоятельной работы
```

Avoid new/delete if possible

Как избежать работы с сырой памятью:

- `std::vector<T>`
- `std::deque<T>`
- `std::list<T>`

- `std::unique_ptr<T>`
- `std::shared_ptr<T>`

Основные конструкции

```
(информация для справки)
```

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

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

Declaration statements

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

Пример:

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

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 ([init-statement] condition) statement

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

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

- `init-statement`:
  - expression statement
  - simple declaration (Click me)

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

Declaration in condition

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

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

```cpp
#include <iostream>
#include <optional>

std::optional<int> compute_result() { return 1; }

int main() {
    if (std::optional<int> value = compute_result()) {
        std::cout << *value;
    }
}
```

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-array 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 следующих лекциях
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

Функция

```cpp
#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;
}
```

Сущность, связывающая последовательность утверждений с именем и набором параметров.

При вызове фунцкии ее параметры инициализируются аргументами, после чего выполняется тел

Вызов функции

В зависимости от соглашении 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
```

Function call operator

Form:

```
F(Arg1, Arg2, ...)
```

- `F` — выражение, результат которого:
  - функция / ссылка на функцию
  - указатель на функцию (см. слайды далее)
  - [вызов метода класса]*
- `Arg1, Arg2, …` — список выражений [или списков инициализации]. Может быть пустым.

Function call operator

Form:

```
F(Arg1, Arg2, ...)
```

Если F - фукнция [или метод], то допускается перегрузка F

```
#include <iostream>

int sum(int i) { return i; }                    // (1)
int sum(int i, int j) { return i + j; }         // (2)

int main() {
    std::cout << sum(1) << std::endl;           // call (1)
    std::cout << sum(1, 2) << std::endl;        // call (2)
}
```

Выбирается та функция, чей набор параметров наиболее подходящий.

Function call operator

Порядок вычисления выражений

Form:

```
F(Arg1, Arg2, ...)
```

`F`, `Arg1`, `Arg2` представляют собой выражения, которые вычисляются:

- В любом порядке(`until C++17`)
- Сначала `F`, затем все остальное в любом порядке(`since C++17`)

Function call operator

Form:

```
F(Arg1, Arg2, ...)
```

Тип выражения — тип, возвращаемый фукнцией.

static in fuction body

```
double someFormula(int i, double x) {
    static const double coef = std::asin(1 / std::sqrt(
    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;
}
```

Способы прерывания функции

- `return` statement
- `throw`-expression

throw, try-catch (basic)

Обработка исключительных ситуаций

Syntax:

```
throw expression;  // (1)
throw;             // (2)
```

```
#include <iostream>
#include <stdexcept> // take error-classes here
void g() {
    throw std::logic_error("msg");
}
void f() {
    try {
        g();
    } catch (const std::logic_error& ex) {
        std::cout << "handle exception: " << ex.what() << std::endl;
        throw; // rethrow
    }
}
```

**Слайд для получения базового представления об исключениях, подробнее рассмотрим эту тему в дальнейших лекциях*

Тип функции

```
void z(int i, double j) {}
int main() {
    z = 10;
    // error: non-object type 'void (int, double)' is not assignable
}
```

- Функция не является объектом
  - нельзя передать по значению в другую функцию
  - нельзя вернуть из другой фукнции
  - нельзя создать массив функций
- Тип функции состоит из типа возвращаемого значения и типов параметров (после array-to-pointer, function-to-pointer преобразований), [`noexcept` (`since c++17`)]*

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
```