

## Структура языка программирования. Программные конструкции.

### 13. Программные конструкции: лямбда-выражение.

**Лямбда-выражение** в программировании – специальный синтаксис для определения функциональных объектов, заимствованный из  $\lambda$ -исчисления, лежит в основе функциональных языков программирования.

Лямбда-выражения поддерживаются во многих языках программирования: Common Lisp, Ruby, Perl, Python, PHP, JavaScript (начиная с ES 2015), C#, F#, Visual Basic .NET, C++, Java, Scala, Kotlin, Object Pascal (Delphi), Haxe (кросс-платформенный язык программирования) и других.

**Анонимная функция** – особый вид функций, которые объявляются в месте использования и не получают уникального идентификатора для доступа к ним.

Обычно при создании анонимные функции

- вызываются напрямую;
- ссылка на функцию присваивается переменной, с помощью которой затем можно косвенно вызывать данную функцию.

В последнем случае анонимная функция получает имя и становится именованной.

Если **анонимная функция** ссылается на переменные, не содержащиеся в её теле (захват переменных), то такая функция называется **замыканием**.

**Лямбда-выражение** — типичная для многих языков синтаксическая конструкция для определения анонимной функции.

Примеры

Язык	Пример записи сложения
<b>C#</b>	<code>(x,y) =&gt; x+y</code>
<b>C++</b>	<code>auto lambda = [](auto x, auto y) {return x + y;};</code>
<b>JavaScript</b>	<code>(x, y) =&gt; x + y;</code> или <code>function(x, y) { return x + y }</code>
<b>Python</b>	<code>lambda x, y: x+y</code>
<b>Go</b>	<code>Z := func() int {     return X + Y }()</code>
<b>Java</b>	<code>(a, b) -&gt; {return a + b;}</code>

## Лямбда-выражения появились в C++11.

Лямбда-выражения представляют собой анонимные функции, которые можно определить в любом месте программы.

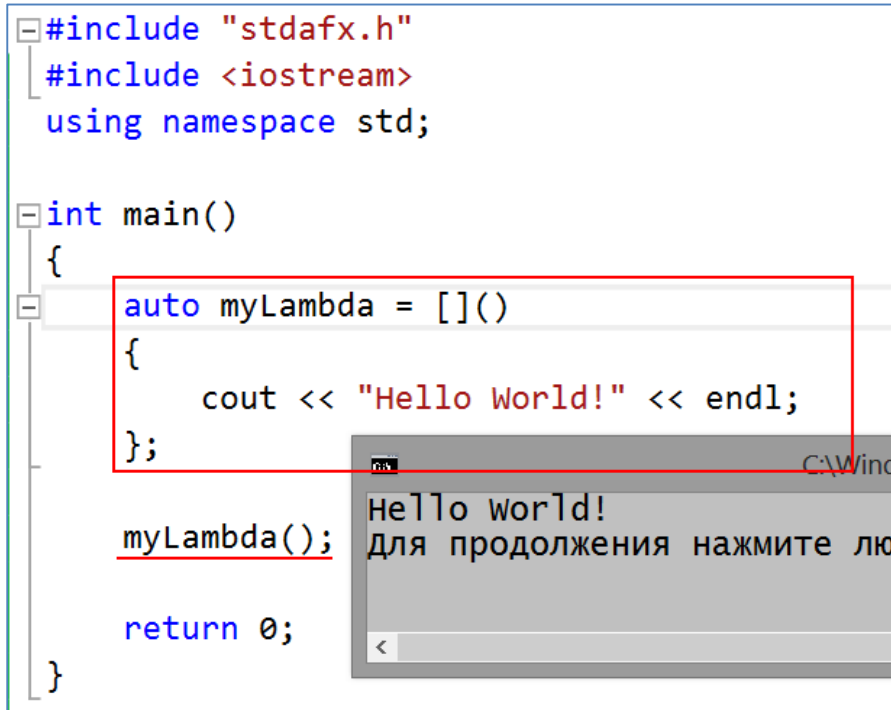
Пример простой лямбда-функции:

```
#include "stdafx.h"
#include <iostream>
using namespace std;

int main()
{
    auto myLambda = []()
    {
        cout << "Hello World!" << endl;
    };

    myLambda();

    return 0;
}
```



где

- |  |   |
|--|---|
| <code>auto myLambda = [](){тело};</code> | - именованное лямбда- выражение<br>( <code>auto</code> - автоматический вывод типа) |
| <code>myLambda ();</code>                | - вызов именованной лямбда-функции  |

## Структура лямбда-выражения:

```
[ < маска_переменных > ] ( < список_параметров > ) mutable throw() ->  
< возвращаемый_тип >  
{  
    /* тело_лямбда-выражения */  
}
```

### Маска переменных (*обязательно*).

Лямбда-выражение может получать доступ к переменным вне лямбда-выражения, определенным в той же области видимости, что и лямбда, и использовать их внутри тела лямбда.

**Маска** определяет способ получения параметров (*захват переменных*) телом лямбда-выражения.

Захват переменных означает, что лямбда может использовать не только переменные, которые передаются в качестве параметров, но и объекты, которые были объявлены вне лямбда-выражения:

- **[a, &b]** **a** захвачена по значению (копия), **b** захвачена по ссылке;
- **[this]** захватывает указатель **this** по значению;
- **[&]** захват всех символов по ссылке (параметр передается по ссылке);
- **[=]** захват всех символов по значению (создается копия значения);
- **[]** ничего не захватывает.

### Список параметров (*необязательно*).

Список параметров лямбда-выражения, аналогичен записи аргументов для обычных функций. Лямбда-выражение может принимать в качестве параметра другое лямбда выражение.

*В C++11 параметры лямбда-выражений требовалось объявлять с указанием конкретных типов.*

*C++14 снимает это ограничение и позволяет объявлять параметры лямбда со спецификатором типа **auto**.*

*Начиная с C++14 можно использовать параметры по умолчанию.*

**mutable** (*необязательно*) — использование **mutable** позволяет модифицировать параметры, переданные по значению.

**throw()** (*необязательно*) — спецификация исключений, то есть лямбда-выражения могут выбрасывать исключения.

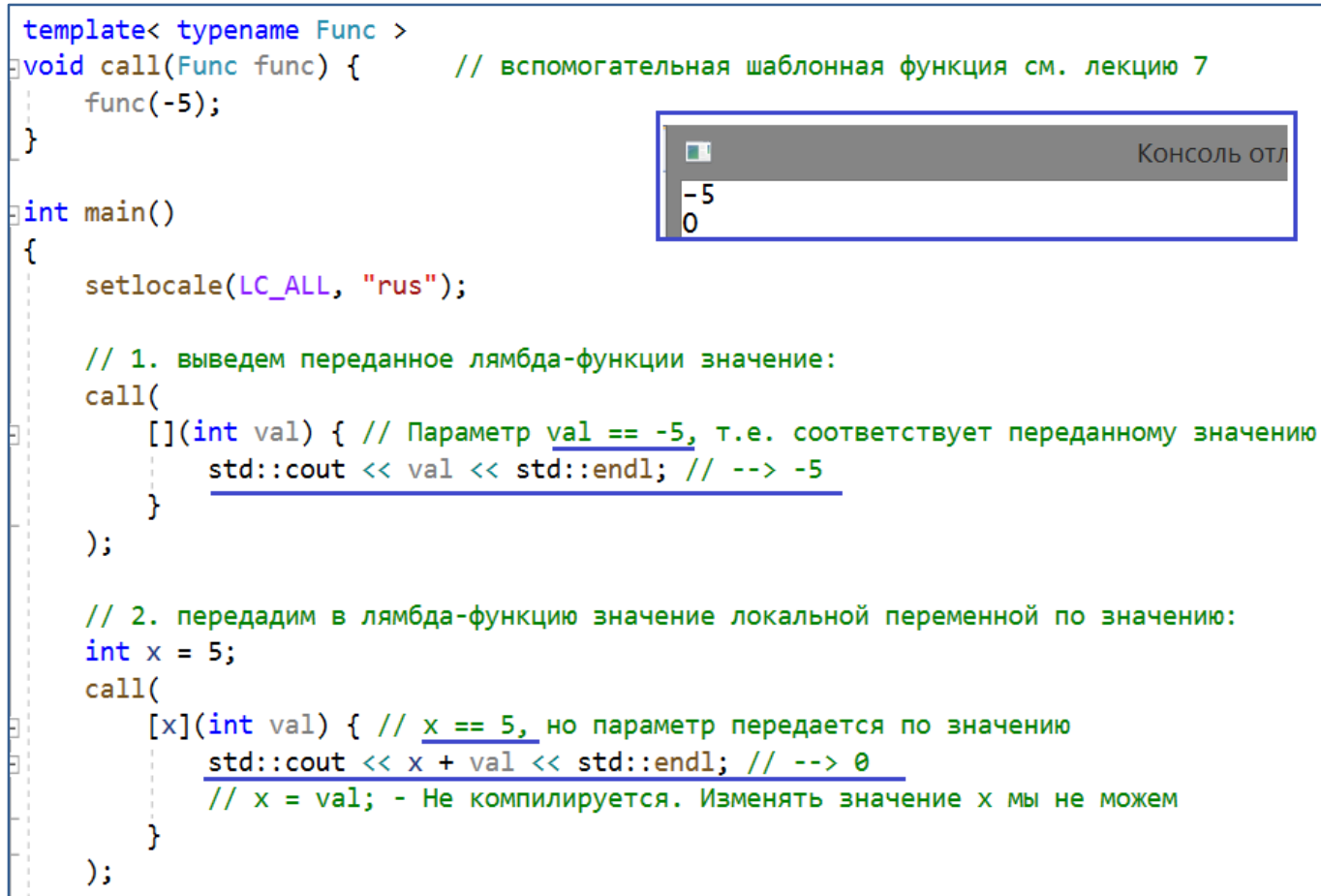
**Возвращаемый тип** — определяет возвращаемый тип лямбда-выражения.

- если у лямбда-выражения нет возвращаемого значения, то по умолчанию устанавливается **void**;

- если в лямбда-выражении есть **return**, то компилятор вычисляет тип возвращаемого значения автоматически на основании **return**-выражения;
- если же в лямбда-выражении присутствует **if** или **switch** и, соответственно, несколько **return**, то требуется явное указание конечного возвращаемого типа в виде: -> **тип\_возвращаемого\_параметра**, перед началом тела функции.

**Тело** лямбда-выражения (*обязательно*).

Пример 1:



```

template< typename Func >
void call(Func func) {           // вспомогательная шаблонная функция см. лекцию 7
    func(-5);
}

int main()
{
    setlocale(LC_ALL, "rus");

    // 1. выведем переданное лямбда-функции значение:
    call(
        [](int val) { // Параметр val == -5, т.е. соответствует переданному значению
            std::cout << val << std::endl; // --> -5
        }
    );

    // 2. передадим в лямбда-функцию значение локальной переменной по значению:
    int x = 5;
    call(
        [x](int val) { // x == 5, но параметр передается по значению
            std::cout << x + val << std::endl; // --> 0
            // x = val; - Не компилируется. Изменять значение x мы не можем
        }
    );
}

```

Консоль отл

-5  
0

Захват переменных:

[] не только определяет (вводит) лямбду, но также содержит список захваченных переменных (может быть пустым). Это называется «список захвата».

Захватив переменную, лямбда создает член-копию этой переменной в типе замыкания. Затем внутри тела лямбды можно получить к этой копии доступ.

```

template< typename Func >
void call(Func func) {      // вспомогательная шаблонная функция см. лекцию 7
    func(-5);
}

int main()
{
    setlocale(LC_ALL, "rus");

    // 3. передадим в лямбда-функцию значение локальной переменной по ссылке:
    x = 33;
    call(
        [&x](int val) { // Теперь x передается по ссылке
            std::cout << x << std::endl; // --> 33
            x = val; // OK!
            std::cout << x << std::endl; // --> -5
        }
    );
    // Значение изменилось:
    std::cout << x << std::endl; // --> -5

    // побочный эффект от связывания переменных с лямбда-функцией по ссылке:
    x = 5;
    auto refLambda = [&x]() { std::cout << x << std::endl; };
    refLambda(); // --> 5
    x = 94;
    // Значение поменялось, что скажется и на лямбда-функции!
    refLambda(); // --> 94
}

```

```

33
-5
-5
5
94

```

```
template< typename Func >
void call(Func func) {      // вспомогательная функция
    func(-5);
}

int main()
{
    setlocale(LC_ALL, "rus");

    // Привязку можно осуществлять по любому числу переменных,
    // комбинируя как передачу по значению, так и по ссылке:
    int y = 55;
    int z = 94;
    int w = 42;

    call(
        [y, &z](int val) {
            std::cout << std::endl << "Привязываем переменную y по значению, z по ссылке" << std::endl;
            std::cout << y << std::endl; // --> 55
            std::cout << z << std::endl; // --> 94
            // y = val; - Нельзя
            z = val; // ОК! значение переменной z изменилось!
        }
    );

    z = 94;
    // привязать сразу все переменные, то можно использовать следующие конструкции:
    call(
        // Привязываем все переменные в области видимости по значению
        [=](int val) {
            std::cout << std::endl << "Привязываем все переменные в области видимости по значению" << std::endl;
            std::cout << y << std::endl; // --> 55
            std::cout << z << std::endl; // --> 94
            std::cout << w << std::endl; // --> 94
        }
    );
};
```

Консоль отладки Microsoft Visual Studio

Привязываем переменную y по значению, z по ссылке  
55  
94

Привязываем все переменные в области видимости по значению  
55  
94  
42

```
template< typename Func >
void call(Func func) {      // вспомогательная функция
    func(-5);
}

int main()
{
    setlocale(LC_ALL, "rus");

    call(
        // Привязываем x по значению, а все остальное по ссылке
        [&, x](int val) {
            std::cout << std::endl << "Привязываем x по значению, а все остальное по ссылке" << std::endl;
            std::cout << y << std::endl; // --> 55
            std::cout << z << std::endl; // --> 94
            std::cout << w << std::endl; // --> 94
        }
    );

    call(
        // Привязываем y по ссылке, а все остальное по значению
        [=, &y](int val) {
            std::cout << std::endl << "Привязываем y по ссылке, а все остальное по значению" << std::endl;
            std::cout << y << std::endl; // --> 55
            std::cout << z << std::endl; // --> 94
            std::cout << w << std::endl; // --> 94
        }
    );
};
```

Консоль отладки Microsoft Visual Studio

Привязываем x по значению, а все остальное по ссылке  
55  
94  
42

Привязываем y по ссылке, а все остальное по значению  
55  
94  
42

## Пример 2:

```
#include <iostream>
using namespace std;

int main()
{
    int m = 1;
    int n = 1;
    cout << "lambda_expression_1" << '\n';
    [](string a) { cout << a << endl << endl; }("Hello!");

    cout << "lambda_expression_2" << '\n';
    [](int a) { cout << a << endl << endl; }(n);

    cout << "lambda_expression_3" << '\n';
    [&, n](int a) mutable {
        m = ++n + a;
        cout << "    lambda -> " << "m=" << m << " n=" << n << " a=" << a << endl;
        cout << m << ' ' << n << ' ' << a << endl;
    }(4);
    cout << "    main -> " << "m=" << m << " n=" << n << endl << endl;

    cout << "lambda_expression_4" << '\n';
    int y = 32;
    auto answer = [y]()
    {
        int x = y + 10;
        return y + x;
    };
    cout << y << endl ;
}
```

lambda\_expression\_1  
Hello!

lambda\_expression\_2  
1

lambda\_expression\_3  
 lambda -> m=6 n=2 a=4  
6 2 4  
 main -> m=6 n=1

lambda\_expression\_4  
32

### **Mutable:**

По умолчанию operator() типа замыкания является константным, и нельзя изменять захваченные переменные внутри тела лямбда-выражения.

Чтобы изменить это поведение, нужно добавить ключевое слово **mutable** после списка параметров.



Большинство функций библиотеки STL принимают *аргумент-предикат*. Такой аргумент позволяет контролировать те или иные аспекты алгоритма.

В примере используется контейнер `vector` (определяет динамический массив). Содержимое контейнера обрабатывается алгоритмами из библиотеки STL, например, алгоритм `count_if` возвращает количество элементов контейнера, удовлетворяющих определенному условию (предикату). В роли предиката может использоваться лямбда-выражение.

Пример 3. Более сложное применение:

```
#include "stdafx.h"
#include <algorithm>
#include <iostream>
#include <vector>

int main()
{
    setlocale(LC_ALL, "Russian");
    std::vector<int> v{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    // определить количество целых чисел в std::vector равных заданному.
    int target1 = 3;
    int num_items1 = std::count(v.begin(), v.end(), target1);
    std::cout << "число: " << target1 << " количество: " << num_items1 << '\n';
    // лямбда-выражение для подсчета элементов, кратных 3.
    int num_items3 = std::count_if(v.begin(), v.end(), [](int i) {return i % 3 == 0; });
    std::cout << "количество элементов, кратных 3: " << num_items3 << '\n';
    std::cout << std::endl << " Лямбда, захват переменных" << std::endl;
    // лямбда-выражение, захват переменных
    for (auto i : v) [i]() {if (i % 3 == 0) std::cout << i << " "; }();

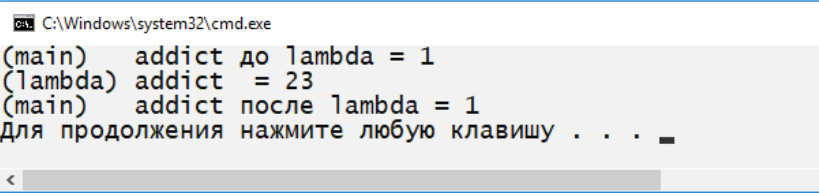
    std::cout << std::endl << " Лямбда с параметрами" << std::endl;
    // Вывод значений кратных 3. Значения будем передавать как параметр у обычной функции
    for (auto i : v) [(auto i) {if (i % 3 == 0) std::cout << i << " "; }(i);

    std::cout << std::endl << " Лямбда без параметров" << std::endl;
    // Вывод значений кратных 3.
    // Значения передаются через захват переменных при этом пропустим скобки для добавления п
    for (auto i : v) [i] {if (i % 3 == 0) std::cout << i << " "; }();
}
```

C:\Windows\system32\cmd.exe  
число: 3 количество: 1  
количество элементов, кратных 3: 3  
Лямбда, захват переменных  
3 6 9  
Лямбда с параметрами  
3 6 9  
Лямбда без параметров  
3 6 9 Для продолжения нажмите любую клавишу . . .

Начиная с C++14 можно использовать параметры по умолчанию:

```
#include "stdafx.h"
#include <iostream>
int main()
{
    setlocale(LC_ALL, "Russian");
    int total = 5;
    int addict = 1;
    auto lam = [&, addict](int coeff = 5)mutable
    {
        total += addict*coeff;
        addict += 22; // переменную можно изменять, но она ограничена телом лямбда-функции
                     // и сохраняется в ней
        std::cout << "(lambda) addict = " << addict << std::endl;
    };
    std::cout << "(main) addict до lambda = " << addict << std::endl;
    lam();
    std::cout << "(main) addict после lambda = " << addict << std::endl;
}
```



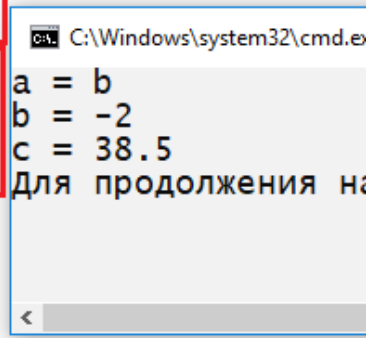
```
C:\Windows\system32\cmd.exe
(main) addict до lambda = 1
(lambda) addict = 23
(main) addict после lambda = 1
Для продолжения нажмите любую клавишу . . .
```

Лямбда-функции можно использовать вместо шаблонных функций:

```
#include "stdafx.h"
#include <iostream>

int main()
{
    auto add = [](auto x, auto y) -> char {
        return x + y;
    };
    auto sub = [](auto x, auto y) {
        return x - y;
    };
    auto mul = [](auto x, auto y) {
        return x * y;
    };

    auto a = add('B', 32);
    auto b = sub(5, 7);
    auto c = mul(5.5, 7);
    std::cout << "a = " << a << std::endl;
    std::cout << "b = " << b << std::endl;
    std::cout << "c = " << c << std::endl;
}
```



```
C:\Windows\system32\cmd.exe
a = b
b = -2
c = 38.5
Для продолжения на
```

## С++20 улучшения лямбда:

1. Предпочтительно использовать синтаксис `[=,this]` как лямбда-захват и исключается неявный захват с использованием синтаксиса `[=]`.

### 2. Шаблонные лямбды

При необходимости определить лямбду, которая будет работать только для одного типа, например, для `std::vector`, на помощь приходят лямбда-шаблоны, где вместо параметра типа можно также использовать концепции, например:

```
auto foo = [] < typename T > (std::vector<T> const & vec) {  
    // делаем специфичные для вектора вещи  
};
```

Раньше в лямбда-функциях было три вида скобок. Теперь их может быть четыре — стало возможным писать явные параметры для лямбда-функций:

1. квадратные скобки для переменных связывания,
2. угловые скобки для шаблонных параметров,
3. круглые скобки для списка аргументов,
4. фигурные скобки для тела функции.

Порядок важен: если его перепутать, будет ошибка. Если шаблонных параметров нет, то угловые скобки не пишутся, потому что пустыми они быть не могут.