Лекция 6

ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ В C++

План

1. Понятие функционального программирования.
2. Лямбда-выражения в C++.
3. Динамическое определение функций с помощью лямбда-выражений.
4. Рекурсия в лямбда-выражениях.
5. Примеры применения лямбда-выражений.

1. Понятие функционального программирования

Функциональное программирование – это парадигма программирования, использующая только композиции функций. Иначе говоря, это программирование в выражениях, а не в императивных командах.

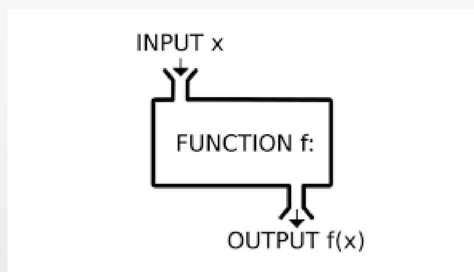
Основными **атрибутами функционального программирования** являются:

- наличие **функций первого класса** (функции наравне с другими объектами можно передавать внутрь функций);
- **рекурсия** является основной управляющей структурой в программе;
- обработка списков (последовательностей);
- запрещение **побочных эффектов** у функций, что в первую очередь означает отсутствие присваивания (в «чистых» функциональных языках);
- отказ от операторов в пользу выражений;
- вместо операторов в идеале вся программа является одним выражением с сопутствующими определениями;
- ключевым вопросом при разработке является «**что нужно вычислить**», а не как;
- использование **функций более высоких порядков** (функции над функциями над функциями).

1. Понятие функционального программирования

В математике функция отображает объекты из одного множества (**множества определения** функции) в другое (**множество значений** функции). Математические функции (их называют **чистыми**) однозначно вычисляют результат по заданным аргументам. Чистые функции не должны хранить в себе какие-либо данные между двумя вызовами. Их можно представлять себе черными ящиками, о которых известно только то, что они делают, но не важно, как.

Программы в функциональном стиле разрабатываются в виде **композиции** функций. При этом функции понимаются почти так же, как и в математике: они отображают одни объекты в другие. В программировании «чистые» функции – идеал, не всегда достижимый на практике. В реальности функции обычно имеют **побочный эффект**: они сохраняют состояние между вызовами или меняют состояние других объектов (например, функции ввода-вывода).



2. Лямбда-выражения в C++

Начиная со стандарта C++11 в языке C++ появились лямбда-выражения (лямбда-функции).

Лямбда-функция – это специальное выражение, создающее так называемое **замыкание**, под которым понимается безымянный объект, который можно вызывать как функцию.

Например:

```
#include <iostream>

int main(void)
{
    std::cout << ([]() -> string {  return "Hello!";  })() << std::endl;
    return 0;
}
```

2. Лямбда-выражения в С++

Лямбда-функция состоит из следующих элементов:

```
[список захвата](параметры)
  mutable           // необязательный
  constexpr         // необязательный
  exceptionattr     // необязательный
  -> тип возврата   // необязательный
{
    тело
}
```

```
  1    2     3         4          5       6
[=]  ()  mutable constexpr  throw() -> int
{                                          (7)
    int n = x + y;

    x = y;
    y = n;
    return n;
}
```

1) **список захвата** – определяет, какие переменные требуется захватить из внешней области видимости («**=**» – захватить все внешние переменные **по значению**, «**&**» – захватить все внешние переменные **по ссылке** и т. д.);
2) **параметры** – определяет список передаваемых в лямбда-функцию параметров;
3) **mutable** – если в лямбда-функции требуется модифицировать захваченные переменные, то ее следует определить как mutable;
4) **constexpr** – результат лямбда-выражения рассчитывается на этапе компиляции;
5) **exceptionattr** – здесь определяется, может ли выражение генерировать исключения;
6) **тип возврата** – задает тип возвращаемого результата;
7) **тело** – программный код, реализующий вычисление лямбда-функции.

3. Динамическое определение функций с помощью лямбда-выражений

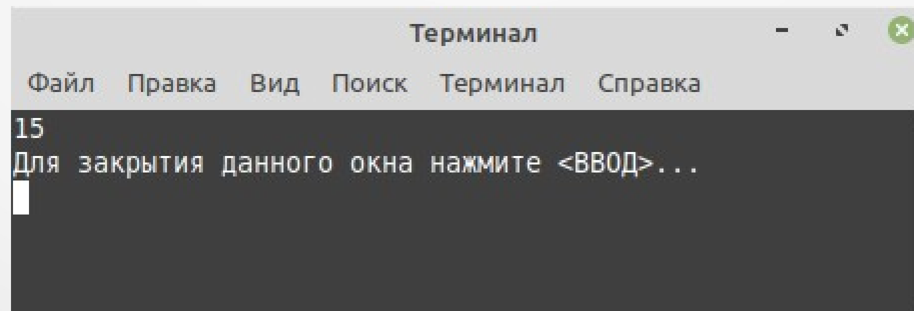
Примеры лямбда-функций

1) Встроенный вызов с захватом всех внешних переменных по значению

```
#include <iostream>

int main(void)
{
    int x = 5, y = 10;

    std::cout << ([=] () -> int { return x + y; })() << std::endl;
    return 0;
}
```



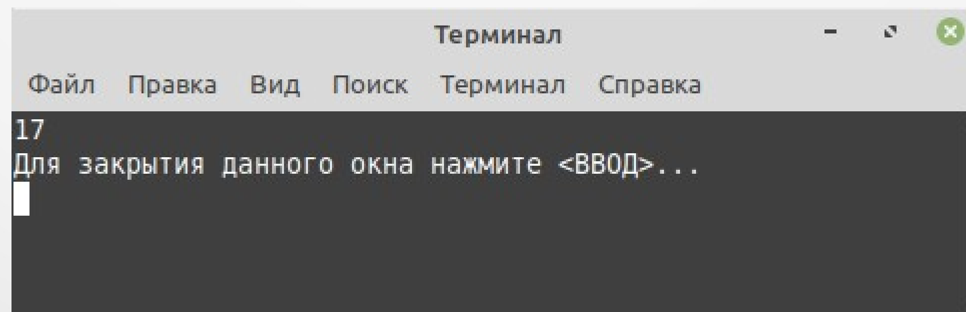
3. Динамическое определение функций с помощью лямбда-выражений

2) Вызов лямбда-функции с захватом всех внешних переменных по ссылке

```
#include <iostream>

int main(void)
{
    int x = 5, y = 10;
    auto sum {[&] () -> int { return ++x + ++y; }};

    std::cout << sum() << std::endl;
    return 0;
}
```



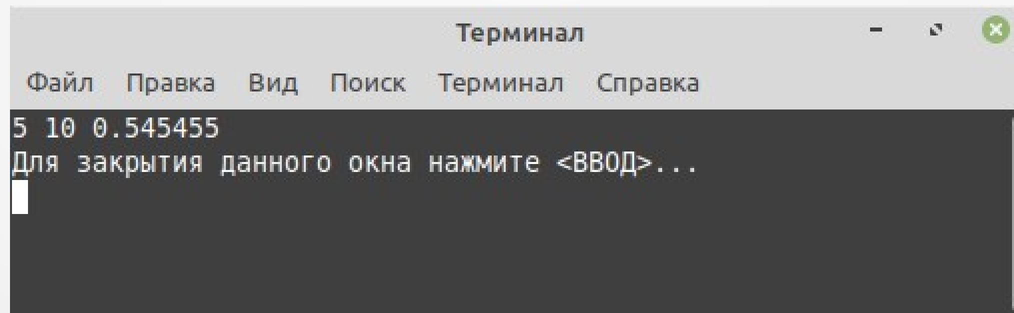
3. Динамическое определение функций с помощью лямбда-выражений

3) Вызов лямбда-функции со списком захватываемых внешних переменных

```
#include <iostream>

int main(void)
{
    int x = 5, y = 10;
    auto fun {[x, y] () mutable -> double { ++x; ++ y; return double(x) / double(y); }};

    std::cout << x << ' ' << y << ' ' << fun() << std::endl;
    return 0;
}
```



Примечание. Если в данном примере опустить квалификатор mutable, компилятор сгенерирует ошибку, т. к. значения внешних переменных x и y изменяются в теле лямбда-выражения.

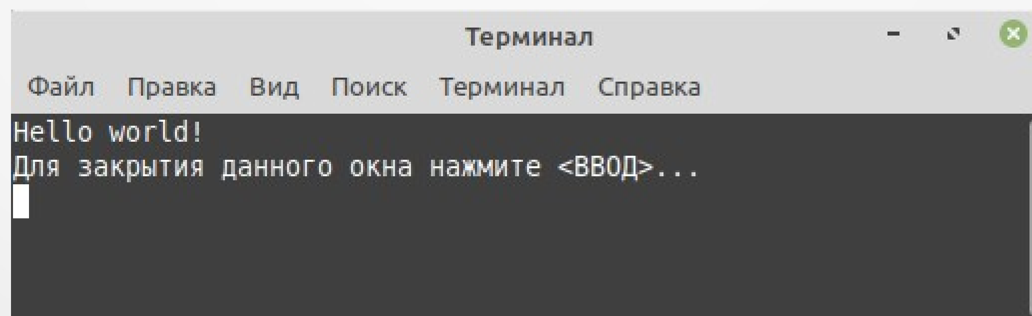
3. Динамическое определение функций с помощью лямбда-выражений

4) Вызов лямбда-функции с параметрами

```
#include <iostream>

int main(void)
{
    auto say {[] (char* str) constexpr -> char* { return str; }};

    std::cout << say((char*)"Hello world!") << std::endl;
    return 0;
}
```

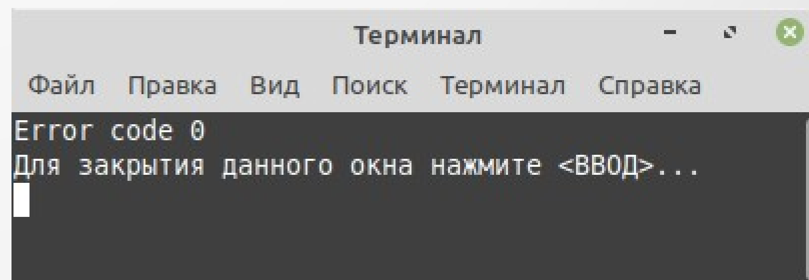


3. Динамическое определение функций с помощью лямбда-выражений

5) Вызов лямбда-функции с обработкой исключений

```
#include <iostream>
int main(void)
{
    auto div {[] (double x, double y) /*noexcept*/ -> double {
                if (y == 0)
                    throw 0;
                return x / y;
            }};

    try {
        std::cout << div(5, 0) << std::endl;
    }
    catch (int e) {
        std::cerr << "Error code " << e << std::endl;
        return 1;
    }
    return 0;
}
```



Примечание: раскомментирование квалификатора noexcept в данном примере приведет к генерации соответствующего предупреждения.

3. Динамическое определение функций с помощью лямбда-выражений

6) Передача лямбда-функций в качестве параметров

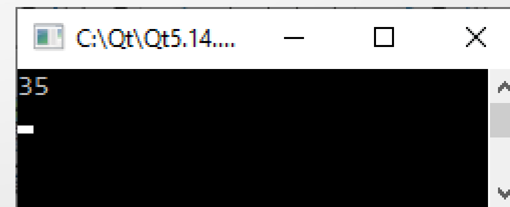
```
#include <iostream>
#include <functional>

using namespace std;

using func = function<double(double)>;

int main()
{
    func left { [](double lhs) { return lhs; } },
         right { [](double lhs) { return lhs; } };
    function<double(func, func, double, double)> mul { [](func lhs, func rhs, double l, double r)
        {
            return lhs(l) * rhs(r);
        }};

    cout << mul(left, right, 10, 3.5) << endl;
    return 0;
}
```



4. Рекурсия в лямбда-выражениях

Для **рекурсивного вызова лямбда-функций** в C++ можно воспользоваться шаблонным классом **std::function**, который по-сути является высокоуровневой оберткой над функциями и функциональными объектами (**функторами**) в C++ (начиная со стандарта C++11). Объекты данного класса могут хранить, копировать и вызывать произвольные вызываемые объекты, например, функции или лямбда-выражения.

В общем виде класс std::function определяется следующим образом:

```
template<class R, class... ArgTypes> class function<R(ArgTypes...)>;
```

Например, определение функтора swap, меняющего местами значения своих аргументов, может выглядеть так:

```
#include <iostream>
#include <functional>

// ...
std::function<void(double&, double&)> swap{[](double& lhs, double& rhs) {
            double tmp = lhs;

            lhs = rhs;
            rhs = tmp;
        }};
// ...
```

4. Рекурсия в лямбда-выражениях

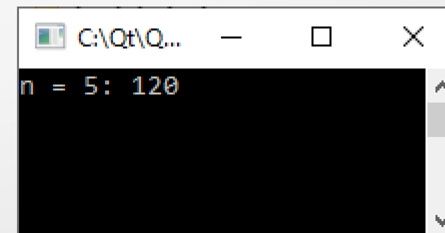
Тогда рекурсивную лямбда-функцию можно на примере вычисления факториала реализовать следующим образом:

```
#include <iostream>
#include <functional>

using namespace std;

int main(void)
{
    function<int(int)> factorial {[&factorial](int n) ->int {
            if (n == 1)
                return 1;
            return n * factorial(n - 1);
        }};

    cout << "n = 5: " << factorial(5) << endl;
    return 0;
}
```



5. Примеры применения лямбда-выражений

1. Кодирование введенного текста с помощью шифра Цезаря.

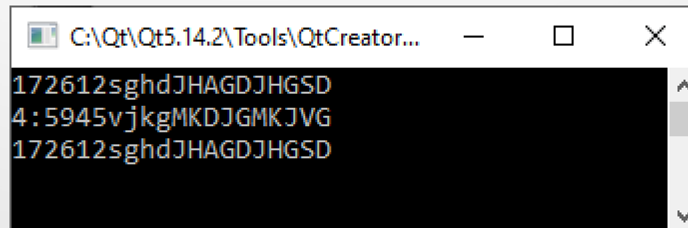
```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    string text;

    cin >> text;
    // Кодирование
    std::transform(begin(text), end(text), begin(text), [](char i) ->char { return i + 3; });
    cout <<  text << endl;
    // Декодирование
    std::transform(begin(text), end(text), begin(text), [](char i) ->char { return i - 3; });
    cout <<  text << endl;

    return 0;
}
```



5. Примеры применения лямбда-выражений

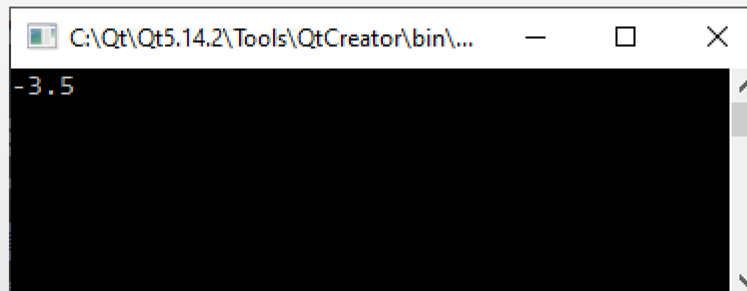
2. Поиск значений по условию в коллекции.

```
#include <iostream>
#include <vector>

using namespace std;

int main(void)
{
    vector<double> arr = { 1, -3.5, -3.14, 2.5 };
    auto result = find_if(arr.begin (), arr.end (), [] (double val) { return val < -3.2; } );

    if (result != end(arr))
        cout << *result << endl;
    else
        cout << "Not found!" << endl;
    return 0;
}
```



5. Примеры применения лямбда-выражений

3. Замена нечетных чисел в массиве на нули.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main(void)
{
    istream_iterator<int> it_cin {cin};
    vector<int> arr;

    copy(it_cin, {}, inserter(arr, arr.begin()));
    transform(arr.begin(), arr.end(), arr.begin(), [](int it) ->int { return (it % 2 == 0) ? it : 0; });
    for (int i : arr)
        cout << i << ' ';
    cout << endl;
    return 0;
}
```