C++ Course 9: Function objects. Lambda expressions.

By Oleksiy Grechnyev

Functions as parameters/variables ?

Функция как параметр/переменная ?

Example 1: A binary operation :

```
int add(int x, int y) {          // Add two int numbers
    return x + y;
}
...
void printOp(??? op) {           // Apply operation op to 7 and 3
    cout << "printOp: op(7, 3) = " << op(7, 3) << endl;
}
...
printOp(add);      // Pass function add as an argument to printOp
??? myOp = add;      // Variable to hold a function
printOp(myOp);      // Pass function myOp as an argument to printOp
```

What type can hold a function? Какой тип может хранить функцию?

Why do we need such things ???

Example 2: Sum a function applied to numbers 1 .. 10:

```
int square(int x) {return x*x;}
int sum_10(??? fun){
	int sum = 0;
	for (int i = 1; i<=10; ++i)
		sum += fun(i);
	return sum;
}
```

Example 3: Error callbacks (Also threads, events, signals, etc.):

```
void errorCallback(const string & s){
	cerr << s;
	exit(1);
}
void setErrorCallback(??? cb) {...}
```

Solution 1: C-style function pointers

Example 1: A binary operation :

```
int add(int x, int y) {return x+y;}
void printOp(int (*op) (int, int)) {...}
int (*myOp) (int, int) = add;     // Variable
printOp(add);     // Pass add to printOp
```

Example 2: Sum a function applied to numbers 1 .. 10:

```
int square(int x) {return x*x;}
int sum_10(int (*fun) (int) ){ ... }
int result = sum_10(square);
```

Example 3: Error callbacks (Also threads, events, signals, etc.):

```
void errorCallback(const string & s){...}
void setErrorCallback(void (*cb)(const string &)) {...}
setErrorCallback(errorCallback);
```

Solution 2: std::function (Usually the best one)

Example 1: A binary operation :

```
int add(int x, int y) {return x+y;}
void printOp(function<int(int, int)> op) {...}
function<int(int, int)> myOp = add;     // Variable
printOp(add);     // Pass add to printOp
```

Example 2: Sum a function applied to numbers 1 .. 10:

```
int square(int x) {return x*x;}
int sum_10(function<int(int)> fun){ ... }
int result = sum_10(square);
```

Example 3: Error callbacks (Also threads, events, signals, etc.):

```
void errorCallback(const string & s){...}
void setErrorCallback(function<void(const string &)> cb) {...}
setErrorCallback(errorCallback);
```

Solution 3: Templates

Example 1: A binary operation :

```
int add(int x, int y) {return x+y;}
template <typename T>
void printOp(T op) {...}
printOp(add);     // Pass add to printOp
```

Example 2: Sum a function applied to numbers 1 .. 10:

```
int square(int x) {return x*x;}
template <typename T> int sum_10(T fun){ ... }
int result = sum_10(square);
```

Example 3: Error callbacks (Also threads, events, signals, etc.):

```
void errorCallback(const string & s){...}
template <typename T>  void setErrorCallback(T cb) {...}
setErrorCallback(errorCallback);
```

Why is std::function better than function pointers?

Почему **std::function** лучше чем указатели на функцию?

1. **std::function** can be used with *functors* and *lambda expressions*.
   std::function может использоваться с *функторами* и *лямбда-выражениями*.

2. **std::function** allows type conversions (допускает преобразования типов)

```
int addRef(const int & x, const int & y) {      // Different signature !
      return x + y;
}

...
function<int(int, int)> myOp = addRef;     // Works !
printOp(addRef);     // Works !
```

Functors

Functor class is a class with **operator()**

Класс-функтор -- это класс с **operator()**

Functor object can be invoked as a function

Класс-функтор может быть вызван как функция

```
struct FunctorAdd {        // struct is a class with default public: access
    int p = 0;             // Parameter
    int operator() (int x, int y) {
        return x + y + p;
    }
};

FunctorAdd fa{17};    // Sets p=17
cout << fa(1, 2);     // Can be called as a function, prints 20
printOp(fa);          // Can be passed to printOp (std::function or template version)
```

Lambda expressions

Lambda expression creates a functor object of an anonymous class :

```
printOp( [](int x, int y)->int{
        return x + y;
} );
```

Use **auto** (or **std::function**) to store it in a variable:

```
auto myOp = [](int x, int y)->int{
        return x + y;
};
printOp(myOp);     // Pass myOp to printOp
```

Lambda expression syntax:

[<capture list>] (<parameters list>) -> <return type> {<body>}

Lambda with **auto** (C++ 14), creates a functor template:

```
[](auto x, auto y) {return x + y;}
```

Example 1 with lambdas

```
void printOp(function<int(int, int)> op) {
	cout << "printOp: op(7, 3) = " << op(7, 3) << endl;
}

int main(){
	printOp( [](int x, int y)->int{
		return x + y;
	} );
	auto myOp = [](int x, int y)->int{
		return x + y;
	};
	printOp(myOp);     // Pass myOp to printOp
}
```

Lambda expression vs Closure

Lambda expression defined an anonymous functor class and creates an object of it.
Лямбда-выражение определяет анонимный функторный класс и создает его объект.

Captured variables are fields of the class.
Захваченные переменные -- поля этого класса.

Lambda expression : Лямбда-выражение :
The expression **[]()->{}** in the code. Выражение **[]()->{}** в коде.

Closure class : Класс замыкания :
The anonymous functor class defined by the lambda expression.
Анонимный функторный класс, определенный лямбда-выражением.

Closure : Замыкание :
The object of this class created by the lambda expression.
Объект этого класса, созданный лямбда-выражением.

Capture. Захват.

Lambdas can capture local variables from the outlying scope:
Лямбды может захватывать локальные переменные из внешней области:

```
int a = 1, b = 2, d = 3, e = 4;
auto myLambda = [a, &b, c = d + e + 18, this] ()->void{
        cout << a << " " << b << " "  << c << endl;
};
```

Capture by value : **a**

Capture by reference : **&b**

Init capture (C++ 14) : **c = a + b + 18**

Class object capture : **this**
For lambdas defined within class methods.

Capture by value

Variables captured by value are *copied when lambda is created*.
Переменные, захваченные по значению, *копируются при создании лямбды*.

```
int a = 13;     // Here a = 13

auto lam = [a]()->void{
        cout << "a = " << a << endl;
};

a = 666;     // Now a = 666

lam();       // What is printed ?
```

Capture by reference

Variables captured by reference are *stored as reference*.
Переменные, захваченные по ссылке, *сохраняются как ссылки*.

```
int a = 13;     // Here a = 13

auto lam = [& a]()->void{
        cout << "a = " << a << endl;
};

a = 666;     // Now a = 666

lam();       // What is printed ?
```

Capture with lambda and functor

```
int p = 3;
```

Capture **p** by a lambda :

```
printOp( [p](int x, int y)->int{
        return x + y + p;
}
```

Capture **p** by a functor :

```
struct {
        int pCap;      // Parameter
        int operator()(int x, int y) {
                return x + y + pCap;
        }
} functor{p};   // pCap captures p by value
printOp(functor);
```

Examples 2, 3 with lambdas

Example 2: Sum a function applied to numbers 1 .. 10:

```
int sum_10(function<int(int)> fun){
        int sum = 0;
        for (int i = 1; i<=10; ++i) sum += fun(i);
        return sum;
}
...
sum_10([](int x) -> int {return x*x;} );
```

Example 3: Error callbacks (Also threads, events, signals, etc.):

```
void setErrorCallback(function<void(const string &)> cb) {...} ...
setErrorCallback([](const string & s) noexcept  ->void{
        cerr << s;
        exit(1);
});
```

Capture this

For lambdas defined inside a class method. Allows the use of class fields in the lambda.

this is captured by value as a *pointer*. The class object itself in *NOT copied* !

```
class MyClass{
    ...
    void run(){
        printOp([this](int x, int y)->int{
                return x + y + p;
        });
    }
    ...
    int p;
};
MyClass mc;
...
mc.run();
```

Init capture or generalized lambda capture (C++ 14)

```
int d = 3, e = 4;
printOp( [p = d*e*2 ](int x, int y)->int{      // By value
        return x + y + p;
});
printOp( [&p = d ](int x, int y)->int{return x + y + p;});       // By reference
```

Init capture can move objects into the lambda :

```
auto ul = make_unique<int>(3);      // A unique_ptr cannot be copied, only moved !
printOp([u = move(ul)](int x, int y)->int{
        return x * y * *u;
});
```

Problem : such lambda does not work with **std::function** !!!

std::function require lambdas to be *copyable* !

Works fine with templates though.

Default capture (Don't do this !!!)

Suppose we have many variables :

int a = 1, b = 2, d = 3, e = 4;

Capture *everything* by value :

lam = [=]()->void{...};

Capture *everything* by reference :

lam = [&]()->void{...};

Class fields are not captured, **this** is !

Reducing the number of arguments

```
int add3(int x, int p, int y) {
        return x + p + y;
}   // One extra argument !
```

How can we use it with **printOp**?

Reducing the number of arguments

```
int add3(int x, int p, int y) {
        return x + p + y;
}   // One extra argument !
```

How can we use it with **printOp**?

With a lambda wrapper (preferred) :

```
printOp([](int x, int y)->int{      // Correct signature !
        return add3(x, 10, y);
});
```

With **std::bind** (returns a functor object) :

```
printOp(bind(add3, placeholders::_1, 10, placeholders::_2));
```

Use lambdas, not **std::bind** !

Using standard operators with std::function

```
void printOp(function<int(int, int)> op) {
	cout << "printOp: op(7, 3) = " << op(7, 3) << endl;
}
```

Can we try it with **operator+** ?

```
printOp(operator+);            // ERROR !!!
printOp(int::operator+);       // ERROR !!!
```

We cannot assign operators to **std::function** !!!
Use the **std::plus<>** wrapper :

```
printOp(plus<int>());          // Works !!!
printOp(plus<>());             // Works in C++ 14 !!!
```

Also : **minus, multiplies, negate, equal_to, less, greater_equal, logical_and, bit_xor, ...**

Using non-static class methods with std::function

```
struct Z{
	int p = 0;
	int op(int x, int y) {
		return x + y + p;
	}
};
Z z{20};
```

Can we assign to an **std::function** ?

```
printOp(z.op);                // ERROR !!!
```

Problem: class methods have an invisible first argument **this**:

```
function<int(Z*, int, int)> funny = Z::op;      // This works !
```

Using non-static class methods with std::function

```
struct Z{
	int p = 0;
	int op(int x, int y) {
		return x + y + p;
	}
};
Z z{20};
```

With a lambda wrapper (preferred) :

```
printOp([&z](int x, int y)->int {      // Correct signature !
	return z.op(x, y);
});
```

With std::bind :

```
printOp(bind(Z::op, &z, placeholders::_1, placeholders::_2));
```

mutable keyword in lambdas

Normally variables captured by value are defined **const** :

```
int a = 10;
auto lam = [a]()->void {
        a += 5;          // ERROR !!!
        cout << a;
};
lam();
```

Use **mutable** modifier !

```
int a = 10;
auto lam = [a]() mutable  ->void {
        a += 5;          // OK !!!
        cout << a;
};
lam();
```

Using lambdas in algorithms:

```
vector <int> v{3, 17, 3, 81, -20, 0, 685, 185, -9, 37, 62};
sort(v.begin(), v.end());    // Sort in ascending order (по возрастанию), uses operator<
// -20 -9 0 3 3 17 37 62 81 185 685
```

How do we sort in *descending* order (по убыванию) ?

Write a lambda:

```
sort(v.begin(), v.end(), [](int x, int y)->bool{
        return x > y;
});
```

Use **std::greater<>** : a wrapper for **operator>**:

```
sort(v.begin(), v.end(), greater<int>());     // C++ 11
sort(v.begin(), v.end(), greater<>());     // C++ 14
```

for_each, count_if, transform

Run a lambda for every element:

```
for_each(v.cbegin(), v.cend(), [](int x)->void{
        cout << x << " : " << 2*x << endl;
});
```

Count negative elements:

```
int n = count_if(v.cbegin(), v.cend(), [](int x)->bool{
        return x < 0;
});
```

Apply a transformation function for each element:

```
transform(v.cbegin(), v.cend(), back_inserter(v2), [](int x)->int{
        return x*2;
});
```

std::back_inserter is a special iterator that inserts (rather than overwrites !)

generate

Generate a sequence of elements:

```
vector<int> v(10);        // Pre-allocate 10 elements
int n = 0;
generate(v.begin(), v.end(), [&n]()->int{
        return n++;
});
// 0 1 2 3 4 5 6 7 8 9
```

The same using number of elements and **std::back_inserter** :

```
vector<int> v;        // No  pre-allocation !
int n = 0;
generate_n(back_inserter(v), 10, [&n]()->int{
        return n++;
});
// 0 1 2 3 4 5 6 7 8 9
```

Thank you for your attention !

title

text