Lambdas: The Functional Programming Companion of Modern C++

Daksh Follow

Feb 16, 2018 · 6 min read

. . .

Lambdas are one of the totally new addition to ***C++11*** and has changed the way we write the code in C++. Any modern C++ code is incomplete without using lambdas which allow creation of in-place anonymous functions and transforms C++ into a full fledged ***Functional Programming*** Language.

> *Since the syntax as well as usage is relatively new, many C++ programmers still find it bit difficult to write and the usage scenarios.*

This blog intends to describe the syntax, usage, and applicabilities of lambdas and how it helps in writing short, maintainable, functional and modern C++ code.

But before understanding what are lambdas and using the same, we need to understand a bit about **Functional Programming**

What is Functional Programming

In a nutshell, **Functional Programming** is all about writing the program functions similar to mathematical function. For example, consider the mathematical function

```
f(x)2 = x * x
```

This function returns the square of `x` and is an independent function which neither depends upon, nor impacts anything outside the function scope. Given an input `x` , the output will always be `x * x` all the time.

Let's also consider polynomial mathematical functions like (x + y)2 (whole square) which is written as

```
f(x,y)2 = x2 + y2 + 2xy
```

This function can simply be broken into multiple smaller functions as

```
f(x,y)2 = f(x)2 + f(y)2 + f(2xy)
```

Here also, the function outputs are dependent only upon the input parameters. With a given input, the function will always generate the same output.

The other characteristics of a function are that the parameters passed in the argument are ***Immutable.*** In both the above functions `f(x)2` and `f(x,y)2` , the paramaters `x and y` are immutable. They are used for calculations, but are not changed because mathematical functions always generate new values out of existing ones. That's the reason in maths, we don't see a function like

```
f(x) = x++
```

But functions which do `x++` is prevalent in C++ and in other programming languages. **Functional Programming** wants us to write code like mathematical functions where the functions are

1. Neither dependent upon, nor change anything which is external to the function.
2. Everything is Immutable, if we change something it will create a new one rather than changing existing items

Why use Functional Programming?

In a world of multi core CPU(s) where the program runs in parallel, a functional program can safely execute of multiple CPU(s) with worrying about excessive locking because it doesn't use or modify any global resources.

> *Functional Programming is not new, just that with the arrival of multi Core CPU(s) it true potential is exposed.*

. . .

The syntax of Lambda

The syntax of lambda consists of `[]` , `()` , and `{}` . This code below is a valid lambda syntax and shall be successfully compiled by the compilers

```
[](){};
```

In this code
- `[]` is Capture list
- `()` is Function Argument provider
- `{}` is lambda Implementation

We can create a very simple lambda which just prints hello world as

```
[](){
 cout<<”Hello World”<<endl;
 };
```

However, the lambda can’t call itself automatically, we need to call them explicitly. We can call the lambda in-place in a similar way we call any other normal functions as

```
[](){
 cout<<”Hello World”<<endl;
}(); // See Here.. we're calling by appending()
```

Lambdas as a function can also return values and the return type of a lambda is depicted using `->` syntax as

```
[]() -> int{
 cout<<”Hello World”<<endl;
 return 1;
}();
```

We can collect the return value of the lambda in any variable as

```
int retLambda = []()->int{
    cout<<"Hello World"<<endl;
    return 1;
}();
```

Just like any other normal functions, we can also pass the parameters in the lambda function inside `()` as

```
int retLambda = [](int value)->int{
   cout<<"Hello World"<<endl;
   return value + 1;
}(100);
```

> *The lambda functions need not be called in-place, we can very well keep the lambda inside a* **Function Pointer** *and call it at a place of our choice.*

We can use C++ `auto` keyword, or use `std::function<>` template to hold the functions, we can also use `C` style *function pointer* `int(*lambdaFn)(int)` for the same, but is generally avoided because of complicated syntax (read Pointers). For example, let's use the `std::function<>` to get the ***Function Reference***

```
// Getting the function reference using std::function<>
std::function<int(int)> lambdaFn = [](int value)->int{
    cout<<"Hello World"<<endl;
    return value + 1;
};

// Calling the lambda function here
int retLambda = lambdaFn(100);
```

The lambda function is just like the mathematical function where it will always return `101` when the value `100` is passed on to it.

We can also use `auto` instead as while taking the function reference

```
auto lambdaFn = [](int value)->int{
    cout<<"Hello World"<<endl;
    return value + 1;
};
```

Creating Lambda for $f(x)^2$ and $f(x,y)^2$

For the function `f(x) = x * x` we can write the lambda functions as

```
std::function<int(int)> fxsquare = [](int x) ->int {
    return x * x;
};
int retValue = fxsquare(10);
```

and for the function `f(x,y)2 = x2 + y2 + 2xy` , we can write the lambda as

```
std::function<int(int, int)> fxsquare = [](int x, int y) -
>int {
    int xsquare = [](int x) { return x * x; }(x);
    int ysquare = [](int y) { return y * y; }(y);
    int twoxy = [](int x, int y) { return 2 * x * y;}(x,y);
    return xsquare + ysquare + twoxy;
};
int retValue = fxsquare(5,3);
```

As you can see with these lambdas that it serves both the purpose of functional programming, i.e it neither depends upon, nor changing anything external to the function as well as treating the input variables as immutable.

We've written a pure functional code here

What are the benefits of this code apart from being functional?

In here, we're not creating multiple functions in global scope which will stay forever. All lambda functions inside are not available outside and shall be out of scope as soon as the function is executed.

At the same time, these lambdas make sure that we're not impacting scoped variables because even if the lambdas are created in place inside a function, it will not be able to access any local variable declared within the scope. For example, in the code below, the variable `localvar` is not accessible inside the lambda function

```
int localvar = 100;
[](){
```

```
    localvar++; // Compiler error..Not Accessible
}();
```

. . .

The Capture List [] of the lambda

The capture list `[]` is used to make local scope variables available to the lambda functions without passing them explicitly inside the parameter argument lists.

The local scope variables can be made available either by value (i.e a copy) or by reference. To pass a value, we need to specify local scope variables in the capture list. Only variables mentioned in the capture list shall be made available to the lambda function as depicted below

```
int localvar = 100;
int localvar2 = 200;
[localvar](){
   cout<<localvar; // Success..
   cout<<localvar2; // Compiler Error... can't access
}();
```

To access all elements, we can either specify all of them in a capture list like

```
int localvar = 100;
int localvar2 = 200;
[localvar, localvar2](){ ... }
```

or we can use `=` in capture list, which means all local scope variables are available inside the lambda functions

```
int localvar = 100;
int localvar2 = 200;
[=](){
   cout<<localvar; // Success...
```

```
    cout<<localvar2; // Success...
}();
```

Pass by value parameters are immutable i.e. we can't change the value of them

```
int localvar = 100;
int localvar2 = 200;
[=](){
   localvar++; // Compiler Error
   localvar2++; // Compiler Error
}();
```

You might have understood that this is done in lambda to enable functional programming immutability, but we can still use C++ reference `&` mechanism to pass local scope variable by reference and change the values. These changes shall be reflected globally.

```
int localvar = 100;
int localvar2 = 200;
[&](){
   localvar++; // Success...
   localvar2++; // Success...
}();
```

Creating Lambda for f(x,y)2 using Capture List

We can use the capture list mechanism to implement the lambda function without passing any parameters as

```
int x = 5, y = 3;
std::function<int(void)> fxsquare = [x,y]() ->int {
    int xsquare = [](int x) { return x * x; }(x);
    int ysquare = [](int y) { return y * y; }(y);
    int twoxy = [](int x, int y) { return 2 * x * y;}(x,y);
    return xsquare + ysquare + twoxy;
};
int retValue = fxsquare();
```

Using Capture List inside the class

Capture lists can also be used inside the class, however, within a class function, only `this` is available, so we can pass only `this` in the capture list to access the class variables inside lambdas

```
template<int iValue, int jValue>
struct AbcTest {
  int i = iValue;
  int j = jValue;
  int sumFn() {
      return [this](){
          return this->i + this->j;
      }();
  }
};
// Using class
AbcTest<100,200> aTest;
int sum = aTest.sumFn();
```

That's all about C++ lambdas. Hope it helps people in understanding as well as encouraging them to use the lambdas within their code

Thanks for reading….!!!!

Daksh