Lambdas in C++

23 Jul 2017

Lambdas vs Functors

As we saw in [last](https://mayankj08.github.io/2017/07/06/Functors-In-C++-II/) post we can use STL functors for common operations like sum, divide. But at times when we need to write our own functor the amount of code needed to define functor is too much. Below is an example to show how useful and concise lambdas can be.

Using functors :

#include<iostream>

#include<vector>

#include<algorithm>

using myPair = std::pair<int,int>;

struct MyCompare{

bool operator()(myPair p1,myPair p2){

return p2.second > p1.second;

}

}mycompare;

int main(){

std::vector<myPair> records(3);

records[0] = std::make\_pair(12,10);

records[1] = std::make\_pair(2,19);

records[2] = std::make\_pair(1,12);

std::sort(records.begin(),records.end(),mycompare);

}

Using lambdas ( >C++11 ):

#include<iostream>

#include<vector>

#include<algorithm>

int main(){

using myPair = std::pair<int,int>;

std::vector<myPair> records(3);

records[0] = std::make\_pair(12,10);

records[1] = std::make\_pair(2,19);

records[2] = std::make\_pair(1,12);

std::sort(records.begin(),records.end(),

[](myPair p1,myPair p2){return p2.second > p1.second;});

}

What is a lambda?

A lambda is syntactic sugar for writing functors. Lambda can be thought of as **unnamed inline function**. Like any another C++ function lambdas has parameters, return type and body. **Unlike a function, lambdas may be defined inside a function.**

Syntax

In general, A lambda expression looks like:

[ capture list ] ( parameter list ) -> return type { function body }

Now let’s try to understand what each entity in above definition means.

[ capture list ]

By default, variables of the enclosing scope cannot be accessed by a lambda.

int a = 10;

auto myLambda = [](){std::cout << a ;}; // ERROR

myLambda();

In above code snippet we are trying to use value of a without capturing, hence error.

The capture list specifies which outside variables are available for the lambda function and whether they should be captured by value or by reference. Capturing a variable makes it accessible inside the lambda.

Capture a by value: [a](){}

Capture a by reference: [&a](){}

When an variable is captured by value we cannot change value of the variable inside lambda.

*/\**

*a is captured by value and we are changing*

*value of a, which is not allowed. So this is*

*an error.*

*\*/*

auto myLambda = [a](){a = a+2;} // ERROR

If we want to change value of variable in lambda we need to capture a by reference (by having & before variable name in capture list). In this case change in value of a inside lambda would be reflected outside lambda too.

int a = 10;

*/\**

*To change value of a in lambda we need*

*to capture by reference, as in below statement.*

*\*/*

auto myLambda2 = [&a](){a = a+2;} // Works!

myLambda2();

print(a); // Value of a becomes 12 outside

// scope of lambda2.

Few points to remember :

* Even if there isn’t any variables to capture, having [] is **mandatory**. We can keep capture list empty like [] when we don’t want to capture anything.
* Don’t confuse capture list with arguments of conventional functions. Unlike function arguments, they do not have to be passed when calling the lambda.
* capture list captures the value of variable when lambda was defined and not at time when lambda was called.
* int a =10;
* auto myLambda = [a](){std:::cout << a;};
* a =14;
* myLambda(); // myLambda get value of a as 10 and not 14.
* Variable should be defined in scope of lambda.
* auto myLambda = [a](){a = a+2;} // Error: a is not defined.
* int a =10;
* myLambda();

( Parameter list )

This is the optional parameters list. We can omit parentheses when we don’t need any argument to lambda (except in case when lambda is defined as mutable, we will see this later).

auto myLambda = [] { std::cout « ” Hello world”; };  
// Valid. Notice we don’t have () of parameter list between capture list and function body.

return type

This is the return type of lambda. Most of the time, compilers can deduce the return type of the lambda expression when you have zero or one return statement, returning same type.

auto myLambda = [](){std::cout << "hi";};

// return type is deduced as void

auto myLambda2 = [](){ if(a==2) return 2; else return 3;};

// return type would be deduced as int

However, in case when multiple return types are of different type compiler won’t be able to deduce the return type and will thrown error.

auto myLambda3 = [](){ if(a==2) return 2; else return 3.5;}; // ERROR

// Compiler won't be able to deduce return type of lambda

// and hence compiler would throw error. We will need to

// specify return type in this case.

auto myLambda4 = []()-> double { if(a==2) return 2; else return 3.5;};

// Works as return type is specified.

This post talks about syntax and some basic rules of lambdas. In next post we will talk about some more concepts in lambdas like mutable lambdas, recursive lambdas etc.