Lambda Functions and Wrappers in C++

Challenge Inside! : Find out where you stand! Try quiz, solve problems & win rewards!

Go to Challenge

Learn via video course

C++ for Beginners

Prateek Narang ₹ Free

5 | *Enrolled: 21330*

Start Learning

Overview

Lambda expressions are inline anonymous functions i.e. they do not have a name. Lambda has various parts such as capture clause, parameters, keywords, body, exceptions, etc. There are three ways in which we can capture external variables to our lambda expression. Lambda and functors are very similar to each other. Lambdas are compact versions of functors. Lambdas are very useful while writing STL algorithms.

Scope

- This article explains Lambda expression in C++, its capture clause, auto and mutual keywords, and its applications.
- This article also explains the Immediate invoke of lambda, lambda vs. functors, and Lambda and STL in C++.

Lambda Function in C++

Lambda function in C++ are in-place functions that can be written for shortcode snippets that are not to be reused so no need to name them as well. It was introduced in Modern C++ beginning from C++11. Writing Lambda expression instead of the function wherever necessary makes the code compact, clean, and easy to understand.

We can define lambda expression locally where we want to pass it to a function as an argument or where we are required to call it. They are anonymous function objects which are used widely for writing in-line functions.

Let us see how we can create and use Lambda Expression

Syntax

```
[Capture clause] (parameters) mutable exception ->return_type
 {
          // Method definition;
 }
```

A lambda consists of the capture clause, parameters, return type, and body of the method.

- **capture clause** - it is a list of variables that are to be copied inside the lambda function in C++. We can also use it to initialize variables.
- **parameters** - zero, one or more than one argument to be passed to the lambda at execution time.
- **mutable** - Mutual is an optional keyword. It lets us modify the value of the variables that are captured by the call-by-value when written in the lambda expression.
- **return type** - It is optional as the compiler evaluates it but in some complex cases compiler can't make out the return type and thus we need to specify it.
- **body of the method** - It is the same as the usual method definition. All the operations to be performed when the lambda expression is called are written here.

Let us see an example of the Lambda Expression

Example:

Let us take an example of a lambda function in C++ to understand the syntax and use. Let us take a list and separate prime numbers from the list. Also, add them to another list using the lambda expression.

```
#include <cmath>
#include<bits/stdc++.h>
using namespace std;

int main()
{
    // list of number
    vector<int> numbers = {137, 171, 429, 467, 909};
    // list for prime numbers
    vector<int> v1 = {};

    // visiting each element of nums and seperating prime number
    // using lambda expression
    for_each(numbers.begin(), numbers.end(),[&v1](int x)mutable{
        bool notPrime = false;
        for(int i=2; i<=sqrt(x);i++){
            if(x%i==0){
                notPrime = true;
                break;
            }
        }
        if(!notPrime) v1.push_back(x);
        });

    cout << "List of prime numbers" << endl;

    for(int i : v1){
        cout << i << " ";
    }
}
```

Output:

```
List of prime numbers
137 467
```

In the example above, we visit each element of the numbers vector. v1 is the list in which we are going to store prime numbers. We are capturing (so that we can use it in our lambda function) v1 by reference and the mutable keyword allows us to edit it as per the program's requirement. If the number is prime it is added to v1 and lastly, we print the components of the list.

Ways to Capture External Variables From Enclosing Scope

Lambda function in C++ is more powerful than the ordinary function. Ordinary functions can only use global variables or the variables passed to the function. Whereas lambda expressions can use local variables present in the main method as well as parameters passed to the lambda expression along with global variables. To use variables other than parameters, we use the capture clause.

There are three ways of capturing these external variables from the enclosing scope:

Capture by Reference

External variables can be captured using their reference to lambda expressions. We pass the address of the variable to the capture clause of the respective expression. This will refer to the original variable and thus changes will also reflect in the original one.

Syntax:

```
[&num1, &num2](){
    // your code here
}
```

Here we are passing the reference of two variables num1 and num2 using the & symbol.

Capture by Value

We can also pass the values of the external variables to the lambda expression. Here we simply pass the variable name to the capture clause. Capturing by value means that we are effectively copying the value of the variable into a new variable that is found inside the lambda expression. The original variable is not affected by the lambda function in C++.

Syntax:

```
[sum](){
    // your code here
}
```

sum must be initialized before using in the capture clause. We are passing the value of the sum to the lambda expression.

Capture by Both (mixed capture)

We can pass more than one variable in the capture clause. Also, not all these variables need to be passed as values or passed by reference. Thus, we can pass both as well as the combination of the values and reference as well.

Syntax:

```
[&id, name](){
    // your code here
}
```

In the above code, we are passing id by reference and name by value. It can have multiple variables in the capture clause and they can be references and/or values.

Example:

```
#include<bits/stdc++.h>
using namespace std;

int main()
{
    // user input for password
    cout << "Enter Passcode" << endl;
```

```
    // actual password
    string password = "me";
    string uspw;
    cin >> userpw;

    // lambda function to check if the password entered is corre
    // capturing both passwords in capture claue
    auto checkPasscode = [userpw, &password](){
        if(userpw.compare(password)==0){
            cout << "Login Successful";
        }else{
            cout << "Incorrect password";
        }
    };
    checkPasscode();
}
```

Output:

```
Enter Passcode
me
Login Successful
```

In the example above, we are passing userpw by value and password by reference. It returns Login Successful as the password matches user input. checkPasscode is a name given to the lambda expression. It stores no value as our lambda expression doesn't return anything.

The Syntax Used for Capturing All Variables

To be able to capture all the variables and use them in our lambda expression we use & and =.

- [&] - It is used to capture all variables by reference. Here we are passing the address of the variable. Thus, if we make any changes to the variable in the lambda expression, it is reflected in the actual variable passed.
- [=] - It is used to capture all variables by value. The changes made to these values are not reflected in the actual variable. Here we are only passing the

value of the variable. The original variable is not affected at all.

Note: when the capture clause is empty, i.e. [], then it can only access variables that are local to it.

Pros & Cons of Capture by Reference

When we capture variables by reference, it uses the address of the original variable, and thus, all the modifications are made on the variable value itself. Whereas, when we pass a value it creates another copy of the variable and modifications take place on it. Also, if a function returns a lambda function, we should not use capture-by-reference as the reference will not be valid.

More on the New Lambda Syntax

Return Values

Return type is mentioned after -> the arrow. If it is not mentioned, the compiler evaluates on its own. In case of complex programs, we are required to specify it. Return value must be returned inside the lambda function if not void.

Throw Specifications

Throw Specifications (also called Exceptions) are optional to mention in lambda function in C++. We can specify which exceptions our lambda expression throws. As we are passing this to another function as an argument, thus this function expects only a certain set of exceptions to be thrown by lambda. It is written after mutable keyword and after parameters in abcense of mutable keyword.

The Auto Keyword

Auto keyword in C++ lets us assign a name to a lambda function. We can call this lambda function using its name just like we call variables by their names. We can also use the auto keyword in passing arguments to the lambda expression. This enables us to perform certain operations on multiple data types. It is also called generic lambda. We use the auto keyword in place of the data type.

For example:

```cpp
#include <iostream>

using namespace std;

int main()
{
    auto add = [](auto a, auto b){
        return a+b;
    };
    cout << add(56,89) << endl;
    cout << add(90.98, 65.42) << endl;
}
```

Output:

```
145
156.4
```

In the above example, add is a lambda expression. auto before add gives a name to the lambda expression. Thus, using this name we can call it multiple times. auto before a and b parameters allows us to pass data of different types. The compiler deduces the type of data based on the data provided.

Initialize Member Variables and Mutual Keyword

In the capture clause, we can create or initialize variables. Also, we can capture existing external variables. To be able to modify any variable that is passed to lambda expression using capture clause, we are supposed to write mutual keyword after parameters list in () parenthesis. It is optional, thus when the capture clause is empty or we don't want to change the variables of the capture clause we can skip writing mutual in the lambda expression.

Let us look at an **example** to understand the concept better

```cpp
#include<bits/stdc++.h>
using namespace std;

int main()
```

```
{
    // lambda expression with auto keyword
    auto code = [value = 10] (int x) mutable{
        x += value;
        value =18;
        cout << "value = " << value << endl;
        cout << "x = " << x << endl;
    };

    code(90);
}
```

Output:

```
value = 18
x = 100
```

In the example above, the value variable is created and initialized in the capture clause. Its value is initialized as 10. We add value to x and change the value to 18. This is possible as we are using the mutable keyword after parameters in lambda expression else it throws an exception. Lastly, we print the values of the value and x variables.

Immediately Invoke a C++ Lambda (IIFE)

Since lambdas are anonymous expressions that are to be used and forgotten. It is a good way to directly execute code without populating the global namespace if the code is not dependent on any conditions.

```
#include<bits/stdc++.h>
using namespace std;

int main()
{
    // details
    string name = "John Williams";
    string college = "St. Xavier's University";
    string batch = "2022";
    char grade = 'A';
```

```
    // lambda expression
    [=](){
       string details = "Student Details\nName: " +name+"\nUnive
       cout << details;
    }();
    // () are must after definition to run the lambda expression
}
```

Output:

```
Student Details
Name: John Williams
University: St. Xavier's University
Batch: 2022
Grade: A
```

In the above example, we have student details that are present in different variables. It is to be grouped in a format and printed. Lambda function is declared, defined, and called using (), right after } (closing curly bracket). Since the function is to be executed only, it is directly called after defining. This is also known as immediately invoking lambda in C++.

> Note: [=] in the above code captures all the external variables by value.

Lambdas vs. Functors

Basically, functors are not functions, they are function objects. Objects of a class in C++ that defines the operator() method (also referred to as call operator or the application operator) are called functors. Objects of this class look like a function as they end with () parenthesis. A functor is advantageous over normal functions as it can support multiple independent states, one for each functor instance, while functions only support a single state.

Lambdas are anonymous function objects that are widely used for writing in-place functions. These are not reused as well as not named.

Let us look at an example to understand the similarities and differences between the two. We will take a functor to encode a given integer value and we will create a lambda function to decode the encoded value.

```
#include<bits/stdc++.h>
using namespace std;

// functor defination
class MyFunctor
{
   public:
     int operator()(int x) { return (x * 2 + 42)/2 - 7;}

};
int main()
{
    MyFunctor encoder;
    int x = encoder(20);
    cout << "Encoded message: " << x << "\n";
    // lambda expression
    auto decoder = [](int x){
        return  ((x+7) * 2 - 42)/2;
    };
    cout << "Decoded message: " << decoder(x);
    return 0;
}
```

Output:

```
Encoded message: 34
Decoded message: 20
```

In the example above MyFunctor is a functor. It is a class that extends operator(), and takes x as a parameter. It performs some operations on it to encode the value and returns it.

To decode the encoded value in x, we use the variable y to store the result. We use a lambda expression for the calculation of the original value.

Both the functor and lambda expressions are very much similar to each other. The only difference is lambda doesn't have a name and its code is in simplified form. We can use functors at multiple places as well as they store state whereas lambdas cannot be used more than once.

Functors are objects of classes so they are a little more complex than lambdas. Also, their implementation lies with the class and it is used in the main() method whereas lambda expression is implemented in the main() method itself.

Lambda and the STL

STL algorithms operate on container elements like vectors, arrays, stacks, etc. using function objects, these function objects are passed as an argument to the algorithms.

For example:

```
#include <bits/stdc++.h>
#include <string>
using namespace std;

int main()
{
    // string array to store colors
    std::string colors[4] = {"Magenta", "Beige", "Cyan", "Lavend

    // for_each loop that checks if the length of the name of th
    // prints the colors that satifies the condition.
    for_each(colors, colors+4, [](std::string i)
    {
        if(i.length()>=5)
            std::cout << i << " ";
    });


    cout << endl;
}
```

Output:

```
Magenta Beige Lavender
```

The above code prints the colors whose length is greater than or equal to 5.

In the example above we are passing an anonymous function object with implementation directly to the STL function. Without Lambda expression, we will have to create a function class, the object of that function class and then we can pass a single line function to the STL. Rather lambda not only makes the code compact but also reduces the complexity of the program with the same functionality.

Applications of Lambdas with Appropriate Example

Applications of lambda expression:

1. Lambda expressions are lightweight, readable, and compact.
2. They improve the locality of the code, functors are written far away from the place they are called.
3. Lambdas allow generic implementation of the function i.e. we can define a general function and use it for multiple data types like int, float, double, etc.
4. Using lambda we can overload the same function multiple times in the main method itself.
5. Lambdas are good at storing the temporary state of a variable.

Let us take an example of a list of employees, we will check the hours they worked a week. If they worked more than 68 hours, then we will add incentives of 20 percent on their salary.

```
#include <iostream>

using namespace std;

int main()
{
    // [i][0] will store id of employee
    // [i][1] will store working hours a week
    // [i][2] will store the salary per month
    int employee[5][3] = {
```

```
        {1001, 63, 25000},
        {1002, 69, 30000},
        {1003, 80, 23000},
        {1004, 73, 40000},
        {1005, 50, 20000},
    };

    // Immediately invoked a C++ lambda
    [&employee]() mutable {
        for(int i = 0; i < 5 ; i++){
            if(employee[i][1]>68){
                employee[i][2]+=(0.2*employee[i][2]);
            }
        }
    }();
     // prinitng final details
    cout << "Employee id\tHours worked\tAmount+Incentive" << end
    for(int i =0;i<5;i++){
        for(int j =0; j<3;j++){
            cout << employee[i][j] << "\t" << "\t";
        }
        cout << endl;
    }
}
```

Output:

```
Employee id     Hours worked    Amount+Incentive
1001            63              25000
1002            69              36000
1003            80              27600
1004            73              48000
1005            50              20000
```

- employee is an array, the first column stores the employee id, the second column stores the number of hours they worked, and the third-row stores the initial salary.

- Using lambda expression we will check the hours worked by each employee and add incentives if the working hours are more than 68 hours a week. We are capturing employee by the reference, the lambda expression doesn't take any parameters and it also doesn't return any value. It makes changes to the employee table thus mutable keyword is used.

Conclusion

- Lambda expressions are short code snippets that are not to be reused and act as an anonymous function.
- Lambda expression has various components such as capture clause, parameters, mutable and auto keywords, exceptions, return type, etc.
- The auto keyword enables us to assign a name to a lambda expression and also to make generic lambda.
- In the capture clause creation and initialization of variables can be done. To edit these variables we use the mutable keyword.
- Lambda expressions and functors are very much similar to each other. Lambdas are created for single-use without a name and in the main method, unlike functors.
- Lambda's are written while implementing STL algorithms in C++. They make the code readable and compact.

Challenge Time!

Time to test your skills and win rewards!

Start Challenge

Note: Rewards will be credited after the next product update.

This article is written by

N Niyati Thakkar

Updated - 28 Jun 2022 • 14 mins read • Published : 28 Jun 2022

C++

How would you rate this article?

Prev

Pair in C++

Next

lvalue and rvalue in C++

Free Courses by top Scaler instructors