

C++ Lambda

In this tutorial, you will learn about C++ lambda expressions with the help of examples.

C++ Lambda expression allows us to define anonymous function objects ([functors](#)) which can either be used inline or passed as an argument.

Lambda expression was introduced in C++11 for creating anonymous functors in a more convenient and concise way.

They are more convenient because we don't need to overload the `()` operator in a separate class or struct.

Creating a Lambda Expression in C++

A basic lambda expression can look something like this:

```
auto greet = []() {  
    // lambda function body  
};
```

Here,

- `[]` is called the **lambda introducer** which denotes the start of the lambda expression
- `()` is called the **parameter list** which is similar to the `()` operator of a normal function

The above code is equivalent to:

```
void greet() {  
    // function body  
}
```

Now, just like the normal functions, we can simply invoke the lambda expression using:

```
greet();
```

Note: We have used the `auto` keyword to automatically deduce the return type for lambda expression.

Example: C++ Lambda Function

```
#include <iostream>
using namespace std;

int main() {

    // create a lambda function that prints "Hello World!"
    auto greet = []() {
        cout << "Hello World!";
    };

    // call lambda function
    greet();

    return 0;
}
```

Run Code >>

Output

```
Hello World!
```

In the above example, we have created a simple program that prints `Hello World!` using a C++ lambda expression.

First, we created the lambda function and assigned it to a variable named `greet`.

```
auto greet = []() {  
    cout << "Hello World!";  
};
```

Then, we have called the lambda function using the `greet` variable along with the `()` operator:

```
// displays "Hello World!"  
greet();
```

C++ Lambda Function With Parameters

Just like a regular function, lambda expressions can also take parameters. For example,

```
#include <iostream>  
using namespace std;  
  
int main() {  
  
    // lambda function that takes two integer  
    // parameters and displays their sum  
    auto add = [] (int a, int b) {  
        cout << "Sum = " << a + b;  
    };  
  
    // call the lambda function  
    add(100, 78);  
  
    return 0;  
}
```

Run Code >>

Output

```
Sum = 178
```

In the above example, we have created a lambda function to which takes two integer parameters and displays their sum.

```
auto add = [] (int a, int b) {  
    cout << "Sum = " << a + b;  
};
```

This is equivalent to:

```
void add(int a, int b) {  
    cout << "Sum = " << a + b;  
}
```

We have then called the lambda function by passing two integer arguments:

```
// returns 178  
add(100, 78);
```

C++ Lambda Function With Return Type

Like with normal functions, C++ lambda expressions can also have a return type.

The compiler can implicitly deduce the return type of the lambda expression based on the `return` statement(s).

```
auto add = [] (int a, int b) {  
    // always returns an 'int'  
    return a + b;  
};
```

In the above case, we have not explicitly defined the return type for the lambda function. This is because there is a single `return` statement which always returns an integer value.


But for multiple `return` statements of different types, we have to explicitly define the type. For example,

```
auto operation = [] (int a, int b, string op) -> double {  
    if (op == "sum") {  
        // returns integer value  
        return a + b;  
    }  
    else {  
        // returns double value  
        return (a + b) / 2.0;  
    }  
};
```

Notice the code `-> double` above. This explicitly defines the return type as `double`, since there are multiple statements which return different types based on the value of `op`.

So no matter what type of value is returned by the various `return` statements, they are all explicitly converted to `double` type.

Example 2: C++ Lambda - Explicit Return Type



```
#include<iostream>
using namespace std;

int main() {

    // lambda function with explicit return type 'double'
    // returns the sum or the average depending on operation
    auto operation = [] (int a, int b, string op) -> double {
        if (op == "sum") {
            return a + b;
        }
        else {
            return (a + b) / 2.0;
        }
    };

    int num1 = 1;
    int num2 = 2;

    // find the sum of num1 and num2
    auto sum = operation(num1, num2, "sum");
    cout << "Sum = " << sum << endl;

    // find the average of num1 and num2
    auto avg = operation(num1, num2, "avg");
    cout << "Average = " << avg;

    return 0;
}
```

[Run Code >>](#)

Output

```
Sum = 3
Average = 1.5
```

In the above example, we have created a lambda function to find either:

- the sum of two integers, or
- the average of two integers

```
auto operation = [] (int a, int b, string op) -> double {  
    if (op == "sum") {  
        // returns an 'int'  
        return a + b;  
    }  
    else {  
        // returns a 'double'  
        return (a + b) / 2.0;  
    }  
};
```

In `main()`, we first find the sum of `num1` and `num2` by passing `"sum"` as the third argument:

```
auto sum = operation(num1, num2, "sum");
```

Here, even though the lambda returns an integer value, it is explicitly converted to `double` type.

Then, we find the average by passing some other string as the argument:

```
auto avg = operation(num1, num2, "avg");
```

C++ Lambda Function Capture Clause

By default, lambda functions cannot access variables of the enclosing function. In order to access those variables, we use the capture clause.

We can capture the variables in two ways:

Capture by Value

This is similar to [calling a function by value](#). Here, the actual value is copied when the lambda is created.

Note: Here, we can only read the variable inside the lambda body but cannot modify it.

A basic lambda expression with capture by value looks as follows:

```
int num_main = 100;

// get access to num_main from the enclosing function
auto my_lambda = [num_main] () {
    cout << num_main;
};
```

Here, `[num_main]` allows the lambda to access the `num_main` variable.

If we remove `num_main` from the capture clause, we will get an error since `num_main` cannot be accessed from the lambda body.

Capture by Reference

This is similar to [calling a function by reference](#) i.e. the lambda has access to the variable address.

Note: Here, we can read the variable as well as modify it inside the lambda body.

A basic lambda expression with capture by reference looks as follows:

```
int num_main = 100;

// access the address of num_main variable
auto my_lambda = [&num_main] () {
    num_main = 900;
};
```

Notice the use of the `&` operator in `[&num_main]`. This indicates that we are capturing the **address** of the `num_main` variable.

Example 3: C++ Lambda Capture by Value



```
#include<iostream>
using namespace std;

int main() {

    int initial_sum = 100;

    // capture initial_sum by value
    auto add_to_sum = [initial_sum] (int num) {
        // here initial_sum = 100 from local scope
        return initial_sum + num;
    };

    int final_sum = add_to_sum(78);
    cout << "100 + 78 = " << final_sum;

    return 0;
}
```

[Run Code >>](#)

Output

```
100 + 78 = 178
```

In the above example, we have created a lambda expression that returns the sum of a local variable named `initial_sum` and an integer parameter `num`.

```
auto add_to_sum = [initial_sum] (int num) {
    return initial_sum + num;
};
```

Here, `[initial_sum]` captures `initial_sum` from the enclosing function by value.

Then, we invoke the function and store its return value in the `final_sum` variable.

```
int final_sum = add_to_sum(78);
```

Inside the lambda function:

- `num` is **78**
- `initial_sum` is **100**

So the result becomes **100 + 78** which is **178**.

Note: Suppose we want to capture multiple variables by value. For example,

```
auto my_lambda = [a, b, c, d, e] () {  
    // lambda body  
}
```

As you can see, this can be a very tedious task. To make our work easier, we can simply use **implicit capture by value**. For example,

```
auto my_lambda = [=] () {  
    // lambda body  
}
```

Here, `[=]` says all the variables of the enclosing function are captured by value.

Example 4: C++ Lambda Capture by Reference



```
#include <iostream>
using namespace std;

int main() {

    int num = 0;

    cout << "Initially, num = " << num << endl;

    // [&num] captures num by reference
    auto increment_by_one = [&num] () {
        cout << "Incrementing num by 1.\n";
        num++;
    };

    // invoke lambda function
    increment_by_one();

    cout << "Now, num = " << num << endl;

    return 0;
}
```

[Run Code >>](#)

Output

```
Initially, num = 0
Incrementing num by 1.
Now, num = 1
```

In the above example, we have created a lambda function that increments the value of a local variable `num` by 1.

```
auto increment_by_one = [&num] () {
    cout << "Incrementing num by 1.\n";
    num++;
};
```

Here `[&num]` is used to capture `num` by reference.

Initially, the value of `num` is 0.

Then, we call the lambda expression `increment_by_one()`. This increases the value of `num` to 1.

Note: To capture all variables of the enclosing function, we can simply use **implicit capture by reference**. For example,

```
auto my_lambda = [&] () {  
    // lambda body  
}
```

Here, `[&]` indicates that all the variables are captured by reference.

Example: C++ Lambda Function as Argument in STL Algorithm

```
#include <iostream>  
#include <vector>  
#include <algorithm>  
  
using namespace std;  
  
int main() {  
    // initialize vector of integers  
    vector<int> nums = {1, 2, 3, 4, 5, 8, 10, 12};  
  
    int even_count = count_if(nums.begin(), nums.end(), [](int num) {  
        return num % 2 == 0;  
    });  
  
    cout << "There are " << even_count << " even numbers."  
    return 0;  
}
```

Run Code >>

Output

There are 5 even numbers.

In the above example, we have used a lambda function in the `count_if` algorithm to count the total even numbers in the `nums` vector:

```
int even_count = count_if(nums.begin(), nums.end(), [](int num) {  
    return num % 2 == 0;  
});
```

Notice that we have provided the lambda expression as the third argument to `count_if`. The lambda expression takes the integer `num` and returns `true` if `num` is even.

Also, we have passed the lambda expression inline.

```
[](int num) {  
    return num % 2 == 0;  
}
```

Frequently Asked Questions

Can we use both capture by value and capture by reference in a single lambda capture clause? >

How can we write a lambda expression without using the `auto` keyword? >

How to use a generic lambda? >

What is mutable keyword in Lambda? >

How to create an immediately invoked lambda expression? >

What is the extended syntax of the C++ Lambda Expression?



Did you find this article helpful?

