

# Lambda Functions

Now, we'll study a special type of function: the lambda.

## WE'LL COVER THE FOLLOWING ^

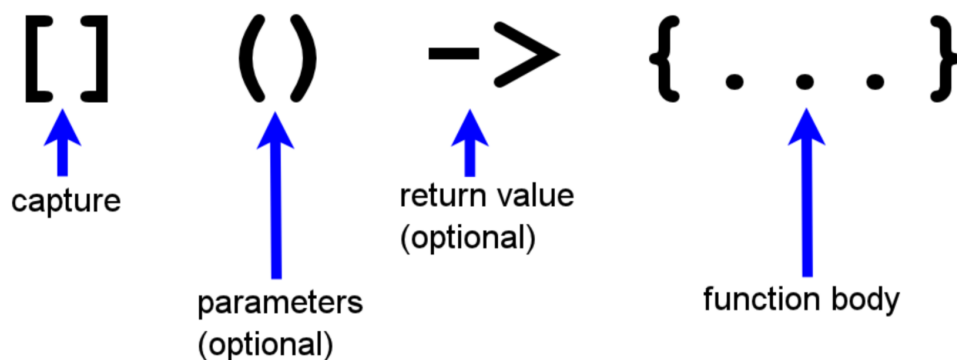
- Syntax
- Function vs. function object
- Closure
- Generic lambda functions
- Capturing local variables

A lambda function, or **lambda**, is a function without a name.

A lambda can be written in-place and doesn't require complete implementation outside the scope of the main program.

A cool feature of lambdas is that they can be treated as data. Hence, they can be stored or copied in variables.

## Syntax #



- `[ ]`: Captures the used variables.
- `( )`: Necessary for parameters

- `()`: Necessary for parameters.
- `->`: Necessary for complex lambda functions.
- `{}`: Function body, per default `const`.
  - `[]() mutable -> {...}` has a non-constant function body.

What exactly do we mean by *capture*?

## Function vs. function object #

The first thing we need to know is that lambdas are just function objects automatically created by the compiler.

A function object is an instance of a class for which the call operator, `operator()`, is overloaded. This means that a function object is an object that behaves like a function. The main difference between a function and a function object is that **a function object is an object and can, therefore, have a state**.

Here is a simple example.

```
int addFunc(int a, int b){ return a + b; }

int main(){

    struct AddObj{
        int operator()(int a, int b) const { return a + b; }
    };

    AddObj addObj;
    addObj(3, 4) == addFunc(3, 4);

}
```

Instances of the struct, `AddObj`, and the function, `addFunc`, are both callable. I just defined the struct `AddObj` in place. That is what the C++ compiler does implicitly if I use a lambda expression.

Have a look.

```
int addFunc(int a, int b){ return a + b; }

int main(){

    auto addObj = [](int a, int b){ return a + b; };

}
```

```
addObj(3, 4) == addFunc(3, 4);
```

```
}
```



That's all! If the lambda expression captures its environment and therefore has a state, the corresponding struct, `AddObj`, gets a constructor for initializing its members. If the lambda expression captures its argument by reference, so does the constructor. The same holds for capturing by value.

## Closure #

Lambda functions can bind their invocation context. This is perhaps the best feature of C++ lambdas.

Binding allows any variables passed in the surrounding scope(invocation context) to be passed to the lambda. This is what the `[]` in the beginning is for. Within these square brackets, we can specify which variables we want the lambda to *capture*.

The empty brackets we've used so far indicate that no variables should be bound.

There are several types of bindings provided by C++ for lambda functions. Have a look:

Binding	Description
<code>[]</code>	no binding
<code>[a]</code>	a per copy
<code>[&amp;a]</code>	a per reference
<code>[=]</code>	all used variables per copy
<code>[&amp;]</code>	all used variables per reference
<code>[=, &amp;a]</code>	per default per copy; a per reference
<code>[&amp;, a]</code>	per default per reference; a per copy
<code>[this]</code>	data and member of the enclosing class per copy
<code>[l= std::move(lock)]</code>	moves <code>lock</code> (C++14)

# Generic lambda functions #

With C++14, we have generic lambdas, which means that lambdas can deduce their argument types. Therefore, we can define a lambda expression such as `[](auto a, auto b){ return a + b; };`. What does that mean for the call operator of `AddObj`?

The call operator becomes a template. I want to emphasize it explicitly: **a generic lambda is a function template**.

Here's an example:

```
#include <iostream>
#include <vector>
#include <numeric>
using namespace std::string_literals;

int main() {
    auto add11=[](int i, int i2){ return i + i2; };
    auto add14= [](auto i, auto i2){ return i + i2; };
    std::vector<int> myVec{1, 2, 3, 4, 5};
    auto res11= std::accumulate(myVec.begin(), myVec.end(), 0, add11);
    auto res14= std::accumulate(myVec.begin(), myVec.end(), 0, add14);

    std::cout << res11 << std::endl;
    std::cout << res14 << std::endl;

    std::vector<std::string> myVecStr{"Hello"s, " World"s};
    auto st= std::accumulate(myVecStr.begin(), myVecStr.end(), ""s, add14);
    std::cout << st << std::endl; // Hello World
}
```



## Capturing local variables #

The difference between the usage of functions and lambda functions boils down to two points:

1. We cannot overload lambdas.
2. A lambda function can capture local variables.

Here is a contrived example of the second point.

```
#include <functional>
```

```

std::function<int(int)> makeLambda(int a){
    return [a](int b){ return a + b; };
}

int main(){

    auto add5 = makeLambda(5);

    auto add10 = makeLambda(10);

    add5(10) == add10(5);

}

```



The function, `makeLambda`, returns a lambda expression. The lambda expression takes an `int` and returns an `int`. This is the type of the polymorph function wrapper, `std::function: std::function<int(int)>`, in line 3.

Invoking `makeLambda(5)` in line 9 creates a lambda expression that captures `a` which is, in this case, is `5`. The same argument holds for `makeLambda(10)` in line 11; therefore, `add5(10)` and `add10(5)` are both `15` in line 13.

Last, here are a couple of tips for how we should design lambdas:

- A lambda should be short and concise.
- A lambda should be self-explanatory, especially since it does not have a name.

---

We will see more examples of lambdas in the next lesson.