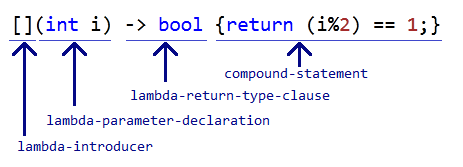
# Lambda functions

Syntax:

1. [ capture-list ] ( params ) { body }
2. [ capture-list ] ( params ) -> return { body }
3. [ capture-list ] { body }



## The capture list

The capture list specifies which outside variables are available for the lambda function. It can be either:

• **[]**: an empty capture list. This implies that no local names from the surrounding context can be used in the lambda body. For such lambda expressions, data is obtained from arguments or from non-local variables.

• **[&]**: implicitly capture by reference. All local names can be used. All local variables are accessed by reference.

• **[=]**: implicitly capture by value. All local names can be used. All names refer to copies of the local variables taken at the point of call of the lambda expression.

If we have for example 2 variables: int x, double y, we might choose how we want to capture them:

[&x, y] – means we capture *x* by reference and *y* by value.

[&] – both variables by reference.

[=] – both variables by value.

Lambdas are useful for example in functions that as an argument, take a comparison function (known as “Comparator”).

For example, std::sort function has this definition:

template< class RandomIt, class Compare >  
void sort( RandomIt first, RandomIt last, Compare comp );

If we have objects of a class Human, and would like to sort them, how would std::sort know how to sort those objects?

We have to give it a function, which will explain how to sort (by age or height etc).

In C++03, it was done by the Functor.

A functor is class\struct which defines the operator()

**C++03 Example**:

Our class that we want to sort:

class Human

{

public:

int age;

int height;

std::string name;

};

struct Human\_Sort\_Function //our Functor

{

bool operator()(const Human &one, const Human &two)

{

return one.age < two.age;

}

};

Usage:

std::vector<Human> humans; //Lets assume we filled vector with 100 humans

Human\_Sort\_Function functor;

std::sort(humans.begin(), humans.end(), functor);

In **C++11** using lambdas this example would look like this:

std::vector<Human> humans; //Lets assume we filled vector with 100 humans

std::sort(humans.begin(), humans.end(),[]( const Human &one, const Human &two) { return one.age < two.age; } );

No functor needed! All in one line! :)

**Lambdas layout in memory:**

Given that we have a function:

void func()

{

int a = 5;

std::string word = "Hi there";

auto our\_lambda = [&word, a](int nr) { return nr + a; };

int result = our\_lambda(10); //result = 15

}

This lambda will be created by compiler like this:

struct Compiler\_Generated\_Lambda

{

std::string &word;

int a;

int operator() (int nr)

{

return nr + a;

}

Compiler\_Generated\_Lambda(std::string &word, int a) : this->word(word), this->a(a) {} //constructor

};

This is how lambdas are created. So for compiler it is the same as functor, but for users, it is a syntactic sugar, that helps with readability and requires less code.

### Conclusion

It’s true that lambdas don’t offer anything you haven’t been able to do before with function objects.

However, lambdas are more convenient than function objects because:

* the tedium of writing boilerplate code for every function class (a constructor, data members and an overloaded operator() among the rest) is relegated to compiler.
* lambdas might be more efficient because the compiler is able to optimize them more aggressively than it would a user-declared function or class.
* lambdas provide a higher level of security because they let you localize (or even hide) functionality from other clients and modules.