

# Functions and Function Objects

Let's take a look at the first two types of callables in C++.

## WE'LL COVER THE FOLLOWING



- Functions
- Function Objects
  - Predefined Function Objects

## Functions #

Functions are the simplest callables. They can have - apart from static variables - no state. Because the definition of a function is often widely separated from its use or even in a different translation unit, the compiler has fewer opportunities to optimize the resulting code.

```
#include <iostream>
#include <vector>
#include <algorithm>

void square(int& i) { i = i * i; }

int main(){
    std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    std::for_each(myVec.begin(), myVec.end(), square);
    for (auto v: myVec) std::cout << v << " ";          // 1 4 9 16 25 36 49 64 81 100

    return 0;
}
```



## Function Objects #

At first, don't call them **functors**. That's a *well-defined* term from the category theory.

**Function objects** are objects that behave like functions. They achieve this due to their call operator being implemented. As functions objects are objects, they can have attributes and therefore state.

```
#include <iostream>
#include <vector>
#include <algorithm>

struct Square{
    void operator()(int& i){i= i*i;}
};

int main(){
    std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    std::for_each(myVec.begin(), myVec.end(), Square());
    for (auto v: myVec) std::cout << v << " "; // 1 4 9 16 25 36 49 64 81 100

    return 0;
}
```



### Instantiate function objects to use them

It's a common error that only the name of the function object ( **Square** ) and not the instance of function object ( **Square()** ) is used in an algorithm: `std::for_each(myVec.begin(), myVec.end(), Square)` . That's of course an error. You have to use the instance:

```
std::for_each(myVec.begin(), myVec.end(), Square())
```

## Predefined Function Objects #

C++ offers a bunch of predefined function objects. They need the header `<functional>` . These predefined function objects are very useful to change the default behaviour of the containers. For example, the keys of the ordered associative containers are by default sorted with the predefined function object `std::less` . But you may want to use `std::greater` instead:

```
std::map<int, std::string> myDefaultMap; // std::less<int>
std::map<int, std::string, std::greater<int> > mySpecialMap; // std::greater<int>
```



There are function objects in the Standard Template Library for arithmetic

There are function objects in the standard Template Library for arithmetic, logic and bitwise operations, and also for negation and comparison.

Function object for	Representative
Negation	<code>std::negate&lt;T&gt;()</code>
Arithmetic	<code>std::plus&lt;T&gt;(), std::minus&lt;T&gt;()</code>  <code>std::multiplies&lt;T&gt;(),</code> <code>std::divides&lt;T&gt;()</code>  <code>std::modulus&lt;T&gt;()</code>
Comparison	<code>std::equal_to&lt;T&gt;(),</code> <code>std::not_equal_to&lt;T&gt;()</code>  <code>std::less&lt;T&gt;(), std::greater&lt;T&gt;()</code>  <code>std::less_equal&lt;T&gt;(),</code> <code>std::greater_equal&lt;T&gt;()</code>
Logical	<code>std::logical_not&lt;T&gt;()</code>  <code>std::logical_and&lt;T&gt;(),</code> <code>std::logical_or&lt;T&gt;()</code>
Bitwise	<code>std::bit_and&lt;T&gt;(), std::bit_or&lt;T&gt;()</code> <code>()</code>  <code>std::bit_xor&lt;T&gt;()</code>

**Predefined function objects**