Functions and Function Objects

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

WE'LL COVER THE FOLLOWING

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;
}
```

The code above takes squares for each of the values in the given set by using `std::for_each` algorithm.

Function objects

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

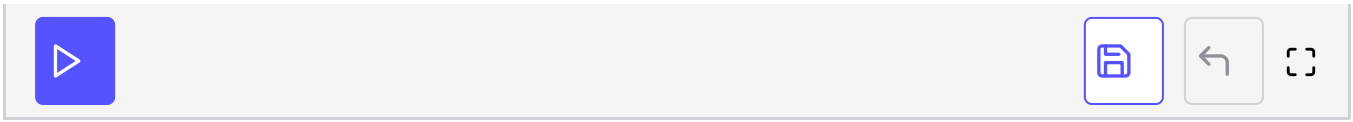
Function objects are objects that behave like functions. They achieve this due to their call operator being implemented. As function objects are objects, they can have attributes and therefore at state.

```
#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. We 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 when changing the default behavior of a container. For example, the keys of the ordered associative containers are, by default, sorted with the predefined function object `std::less` . But we may want to use `std::greater` instead:

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

```
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, logic and bitwise operations, and also for negation and comparison.

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

Predefined function objects

Further information

- functors
- Function objects

In the next lesson, we'll discuss a very important type of function: lambda functions.