# - Examples

Let's understand lambdas better by looking at a few examples.

**WE'LL COVER THE FOLLOWING**


- Lambdas with a vector
  - Explanation
- Closure with lambdas
  - Explanation
- `this` binding
  - Explanation
- A generic lambda
  - Explanation


## Lambdas with a vector

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

bool lessLength(const std::string& f, const std::string& s){
  return f.length() < s.length();
}

class GreaterLength{
  public:
    bool operator()(const std::string& f, const std::string& s) const{
      return f.length() > s.length();
    }
};

int main(){

  // initializing with a initializer lists
  std::vector<std::string> myStrVec = {"12345", "123456", "1234", "1", "12", "123", "12345"};
  std::cout << "\n";
```

```
  // sorting with the function
  std::sort(myStrVec.begin(), myStrVec.end(), lessLength);
  std::copy(myStrVec.begin(), myStrVec.end(), std::ostream_iterator<std::string>(std::cout, '

  std::cout << "\n";

  // sorting in reverse with the function object
  std::sort(myStrVec.begin(), myStrVec.end(), GreaterLength());
  std::copy(myStrVec.begin(), myStrVec.end(), std::ostream_iterator<std::string>(std::cout, '
  std::cout << "\n";

  // sorting with the lambda function
  std::sort(myStrVec.begin(), myStrVec.end(), [](const std::string& f, const std::string& s){
  std::copy(myStrVec.begin(), myStrVec.end(), std::ostream_iterator<std::string>(std::cout,
  std::cout << "\n";

  // using the lambda function for output
  std::for_each(myStrVec.begin(),  myStrVec.end(), [](const std::string& s){std::cout << s <<

  std::cout << "\n\n";

}
```

## Explanation

- We have created a `lessLength()` function on line 7 that returns `true` if the first string is smaller than the second one in length.
- This function can be used as the sorting criteria for `std::sort` on line 25. However, the lambda on line 35 performs the same task in a simpler way.
- As we can see, the parameters of the lambda are the same as those of the defined function.
- In line 30, the `operator()` method of the `GreaterLength()` class is being used to sort the vector in descending order. However, this could also have been done using a lambda similar to the one on line 35.

# Closure with lambdas

```
#include <iostream>
#include <string>

int main(){
  std::cout << std::endl;

  std::string copy = "original";
  std::string ref = "original";
  auto lambda = [copy, &ref]{std::cout << copy << " " << ref << std::endl;};
  lambda();
```

```
  lambda();
  copy = "changed";
  ref = "changed";

  lambda();

  std::cout << std::endl;
}
```

## Explanation

This is a great example of how variables from the environment can be accessed in a lambda.

- The `copy` variable is captured as a copy. Hence, its value is simply copied and bound to the lambda.
- In line 11, when the value of `copy` is altered to `"changed"`, the change isn't reflected in the `lambda()` call. This is because `lambda()` is bound to the original value of `copy`.
- For `ref`, there is the opposite effect, since it is captured as a reference. A change in its value on line 12 is reflected in `lambda()` as well.

# `this` binding

```
#include <iostream>

class ClassMember{
  const static int a = 1;
  int get10(){
    return 10;
  }
  public:
    void showAll(){
      // define and invoke (trailing () ) the lambda functions
      [this]{std::cout << "by this = " << get10() + a  << std::endl;}();
      [&]{std::cout << "by reference = " << get10()+ a  << std::endl;}();
      [=]{std::cout << "by copy = " << get10() + a << std::endl;}();
    }
};

int main(){

  std::cout << std::endl;

  ClassMember cM;
  cM.showAll();

  std::cout << std::endl;
```

```
  std::cout << std::endl;

}
```

## Explanation

This is an example of how the `this` binding works with lambdas.

- `this` binds all the members of a class to the lambda. It is very similar to the `=` binding.

# A generic lambda

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

auto add = [](int i, int i2){ return i + i2; };
auto add14 = [](auto i, auto i2){ return i + i2; };

int main(){

  std::cout << std::endl;

  std::cout << "add(2000, 11): " << add(2000, 11) << std::endl;

  std::cout << "add14(2000, 14): " << add14(2000, 14) << std::endl;
  std::cout << "add14(2000L, 14): " << add14(2000L, 14) << std::endl;
  std::cout << "add14(3, 0.1415): " << add14(3, 0.1415) << std::endl;
  std::cout << "add14(std::string(Hello), std::string(World)): "
            << add14(std::string("Hello "), std::string("World")) << std::endl;

  std::cout << std::endl;

  std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
  auto res = std::accumulate(myVec.begin(), myVec.end(), 1, [](int i, int i2){ return i * i2;
  std::cout << "res: " << res << std::endl;

  auto res14 = std::accumulate(myVec.begin(), myVec.end(), 1, [](auto i, auto i2){ return i *
  std::cout << "res14: " << res14 << std::endl;

  std::cout << std::endl;

}
```

## Explanation

- The `add` lambda on line 7 works solely with `int` arguments.
- We can make it generic by using the `auto` type, as done on line 8.
- Now, any two types that support the `+` operation with each other will automatically work in `add14`.
- Lines 17 and 18 show this generic lambda in action. Floats, integers and long integers can all be added together. The compiler will decide the return type of the function.
- A generic function is also being used to calculate the cumulative product of `myVec` on line 28. There's nothing tricky going on here.

---

That brings us to the end of our discussion on functions.

Try the exercises in the next lesson to gain a better understanding of lambdas.