Course
ELEC-A7151
Course materials
Your points
Code
Code Vault
Course Pages
MyCourses
Teams Channel

This course has already ended.
The latest instance of the course can be found at: Object oriented programming with C++: 2023 Autumn

« 2 Exception handling Course materials 4 Learning environment survey »
ELEC-A7151 / Module 5: Advanced Topics / 3 Lambda expressions

3 Lambda expressions¶

- **You can download the template for the programming tasks of the module as a `zip` file from `this link`.**

Contents

C++ Primer Chapter 10.3 (Customizing operations)

Module 2 discussed algorithms and function parameters that defined the algorithm behavior. Often these functions are pretty small, not uncommonly one-liners that specify some criteria for the algorithm.

Lambda is an expression that defines an anonymous callable function. Lambda expressions were introduced in C++11, and they are useful with algorithms, particularly if the specified behavior is not needed in more than one or two places in the code. To illustrate how lambdas work, below is a sorting algorithm for strings.

```
 1#include <vector>
 2#include <iostream>
 3#include <algorithm>
 4
 5int main() {
 6    std::vector<std::string> starks = {"Bran","Arya","Sansa","Rickon","Robb"};
 7
 8    // Sort the vector based on string lengths using sort function.
 9    // Comparison function is given as part of the lambda expression
10    sort(starks.begin(), starks.end(),
11        [](const std::string& a, const std::string& b) {
12            return a.size() < b.size();
13        });
14
15    // Go through each member of the sorted vector using for_each function.
16    // Lambda function definition outputs each member of the vector to stdout.
17    for_each(starks.begin(), starks.end(),
18        [](const std::string& s) {
19            std::cout << s << " ";
20        });
21
22    std::cout << std::endl;
23}
```

In Section 2 we had an example on algorithms that used the *Shorter* function as a parameter to the sort function, to arrange a given vector of strings based on their length.

With lambdas, we do not need a separate named function for that. In the above program the functionality of the *Shorter* function is included as a **lambda expression** on lines 11-13 and 18-20. Lambda expression includes a **capture list** in brackets (`[]`, in this case empty), **parameter list** (as in functions), and the body of the lambda function:

```
[/*capture list*/](/*parameter list*/) { /*function body*/ }
```

Like normal functions, a lambda function can also have a return value. In this case the type of the return value is determined automatically, but it can specified also explicitly, using the **trailing return** syntax:

```
[...](...) -> /*return type*/ { ... }
```

To illustrate lambdas, here is a variant of the above *sort* call that explicitly defines the return type as bool:

```
sort(starks.begin(), starks.end(),
    [](const std::string& a, const std::string& b) -> bool {
        return a.size() < b.size();
    });
```

In straight-forward cases the compiler will be able to determine the correct return value type automatically (as with the *auto* keyword).

The capture list can be used to reflect variables from the calling program into the lambda function. To illustrate how the capture list works with lambda expressions, here is another example based on find_if algorithm, that relays the variable *len* into the lambda function:

```
 1#include <vector>
 2#include <string>
 3#include <iostream>
 4#include <algorithm>
 5
 6int main() {
 7    std::vector<std::string> starks = {"Bran","Arya","Sansa","Rickon","Robb"};
 8
 9    unsigned int len = 5;
10
11    // variable len will be used inside the lambda function
12    auto it = find_if(starks.begin(), starks.end(),
13            [len](const std::string& a)
14            { return len <= a.size(); });
15
16    std::cout << *it << std::endl;
17}
```

The *find_if* function will return an iterator to the first element in between the given iterators that meets the given criteria. Here we want to return an element that is at least *len* characters long. The capture list provides a way to transfer variables from the outside scope to within the lambda function, as done here for variable *len*. We could not use an additional function argument for this, because then the function interface would not be compatible with what *find_if* expects (an function with one argument). Also, multiple variables can be included in a capture list, separated by commas.

We can also assign lambdas to variables:

```
auto greet =
    [](const std::string& name) {
        std::cout << "Hello, " << name << "!" << std::endl;
    };
```

After which we can use the lambda like any other function using the variable:

```
greet("Erkki");
greet("Kaisa");
```

```
 1#include <string>
 2#include <iostream>
 3
 4int main() {
 5    // assign a lambda to a variable
 6    auto greet =
 7        [](const std::string& name) {
 8            std::cout << "Hello, " << name << "!" << std::endl;
 9        };
10
11    // call the lambda function
12    greet("Erkki");
13    greet("Kaisa");
14}
```

3.1 Different types of captures¶

Like function arguments, the capture variables in lambdas can be passed by value (copied) or by reference. By default, the variable is passed by value, i.e., any modifications on the variable done inside the lambda function do not affect the value of the variable outside the lambda. The variable can also be defined to be passed **by reference**:

```
[&variable](...) { ... };
```

in which case the changes to *variable* inside the lambda function will also affect its value outside the lambda.

Implicit capture lists do not specify the capture variables explicitly, but the compiler determines when an outside variable will be used inside lambda function. Format `[=](...) { code };` tells the compiler to automatically include the variables "code" uses following the *by value* semantic. On the other hand, `[&](...) { code };` is an implicit capture using the *by reference* semantic.

Below example is a slight modification of the above, this time using implicit by value capture list.

```
 1#include <vector>
 2#include <string>
 3#include <iostream>
 4#include <algorithm>
 5
 6int main() {
 7    std::vector<std::string> starks = {"Bran","Arya","Sansa","Rickon","Robb"};
 8
 9    unsigned int len = 5;
10
11    auto it = find_if(starks.begin(), starks.end(),
12        [=](const std::string& a) {
13            return len <= a.size();
14        });
15
16    std::cout << *it << std::endl;
17}
```

3.2 Programming task¶

Points 20 / 20 My submissions 19
Deadline Friday, 22 October 2021, 19:59
To be submitted alone

This course has been archived (Saturday, 17 December 2022, 19:59).

Transcript

Objective: Trying lambdas in action and more practice with algorithms and iterators and trying.

Transcript is a structure that holds the course records for a particular student. For each student there is a list of course/grade pairs for all the courses the student has passed. Again, **transcript.hpp** contains the description of the class that you will need to implement.

Implement first the functions **Add** and **RemoveById** that will add and remove entries from the registry. RemoveById should remove all entries with specific student ID. Hint: remove_if.

Then implement functions **FindTranscript** and **FindCourseResults**:

- **FindTranscript** should find entries based on given student ID
- **FindCourseResults** should return a list of student ID and grade pairs corresponding to a course name.

These functions will not require many lines of code if you use algorithms efficiently.

transcript.cpp

Choose File No file chosen

transcript.hpp

Choose File No file chosen

Submit

« 2 Exception handling Course materials 4 Learning environment survey »