



# Standard Library Algorithms

## Chapter 16

# Today we will discuss



- Minimum Iterator Requirements
- Lambda Expressions
- Algorithms
- Function objects

# Minimum Iterator Requirements

The *Standard Library* separates algorithms from containers, with few exceptions.

The separation allows to add new algorithms and to use them on different containers.

# Minimum Iterator Requirements



The *Standard Library* separates algorithms from containers, with few exceptions.

The separation allows to add new algorithms and to use them on different containers.

We discussed in the previous class that iterators are implemented for each type of the container. The type of the iterator defines which algorithms can be applied to the container.

If an algorithm requires *forward iterator*, then that algorithm will only be able to operate on the containers that support *forward iterators*, *bidirectional iterators*, or *random-access iterators*.

# Minimum Iterator Requirements

## *Iterator invalidation*

Since iterators simply point to container elements, it is possible for them to become *invalid* when a certain container modification occurs.

For example, if we call the `clear()` on a `vector`, all its elements will be destroyed and any iterators that were pointing to elements will be *invalid* now.



# Minimum Iterator Requirements

*Iterator invalidation when inserting into a/an:*

- **vector**  
*all iterators are invalidated if the vector is reallocated;  
if no reallocation happens, the iterators from the insertion  
point to the end of the vector are invalidated*
- **deque**  
*all iterators are invalidated*
- **list or forward\_list**  
*all iterators remain valid*
- **ordered associative container**  
*all iterators remain valid*
- **unordered associative container**  
*all iterators are invalidated if the container is reallocated*

# Minimum Iterator Requirements



When erasing from a container, iterators to the erased elements are invalidated.

In addition, for a

- **vector**

*the iterators from the erased element to the end of the vector are invalidated*

- **deque**

*if an element in the middle of the deque is erased, all iterators are invalidated*

# Algorithm `for_each`

defined in header `<algorithm>`

```
template< class InputIt, class UnaryFunction >  
UnaryFunction for_each( InputIt first, InputIt  
last, UnaryFunction f );  
(until C++20)
```

```
template< class InputIt, class UnaryFunction >  
constexpr UnaryFunction for_each( InputIt first,  
InputIt last, UnaryFunction f );  
(since C++20)
```

```
template< class ExecutionPolicy, class ForwardIt,  
class UnaryFunction2 >  
void for_each( ExecutionPolicy&& policy, ForwardIt  
first, ForwardIt last, UnaryFunction2 f );  
(since C++17)
```

Template algorithm `for_each` calls a function to perform a task once for each element of the container.



# Algorithm `for_each`

defined in header `<algorithm>`

Template algorithm `for_each` calls a function to perform a task once for each element of the container.

```
for_each(myV.begin(), myV.end(), ... )
```



# Lambda Expressions

Many *Standard Library* algorithms may receive function pointers as parameters



# Lambda Expressions

Many *Standard Library* algorithms may receive function pointers as parameters, since function name is easily convertible to a pointer to that function's code.

Before we can pass a function pointer to an algorithm, a corresponding function must be declared.

# Lambda Expressions



Many *Standard Library* algorithms may receive function pointers as parameters, since function name is easily convertible to a pointer to that function's code.

Before we can pass a function pointer to an algorithm, a corresponding function must be declared.

Starting from C++ 11 there is a convenient shorthand notation, using *lambda expressions*, for creating functions without names, called *anonymous functions*.

*Lambda expressions* can be used instead of the *function pointers*.

# Lambda Expressions

**Lambda expressions** or simply **lambdas** are defined locally inside the functions and can use and manipulate local variables of the enclosing function.





# Lambda Expressions



Lambda expressions or simply lambdas are defined locally inside the functions and can use and manipulate local variables of the enclosing function.

## Syntax:

[captures] <tparams> (params) specifiers  
exception attr -> ret requires {body}

The one we will mostly use today:

[captures] (params) {body}

# Lambda Expressions



Lambda expressions or simply **lambdas** are defined locally inside the functions and can use and manipulate local variables of the enclosing function.

## Syntax:

[captures] (params) {body}

[ ] is the *lambda introducer*

**captures** is a comma separated list of captures, i.e. local variables of the function from which the lambda is defined; it can be empty

**params** is the list of parameters; if **auto** is used as a type of a parameter, its type is to be “*inferred*” by the compiler (since C++14, called **generic lambdas**)

# Lambda Expressions



Lambda expressions or simply **lambdas** are defined locally inside the functions and can use and manipulate local variables of the enclosing function.

## Syntax:

[captures] (params) {body}

**Examples:** assume **myV** is a vector of integer values

```
for_each(myV.begin(), myV.end(),  
        [](auto item) {cout << item << endl; });
```

*no local variables are used*



# Lambda Expressions



Lambda expressions or simply **lambdas** are defined locally inside the functions and can use and manipulate local variables of the enclosing function.

## Syntax:

[captures] (params) {body}

**Examples:** assume **myV** is a vector of integer values, and **sum** is a local variable initialized to 0.

```
for_each(myV.begin(), myV.end(),  
         [&sum](auto item) { sum+= item; });
```

*local variable is captured by reference*



# Lambda Expressions



Lambda expressions or simply **lambdas** are defined locally inside the functions and can use and manipulate local variables of the enclosing function.

## Syntax:

**[captures] (params) {body}**

let's see **lambdasExamples.cpp** for some work with lambda expressions



# Lambda Expressions



Lambda expressions or simply lambdas are defined locally inside the functions and can use and manipulate local variables of the enclosing function.

## Syntax:

```
[captures] <tparams> (params) exception ->  
ret requires {body}
```

**tparams** is an optional template parameter list (since C++ 20), that can be followed by

**requires** clause that specifies the constraints on the template parameters (since C++ 20)

# Lambda Expressions



Lambda expressions or simply lambdas are defined locally inside the functions and can use and manipulate local variables of the enclosing function.

## Syntax:

```
[captures] <tparams> (params) exception ->  
ret requires {body}
```

exception provides the list of exceptions that might be directly or indirectly thrown

ret is return type; if not present it is implied by the function return statements, or void if it doesn't return any value

# Algorithms

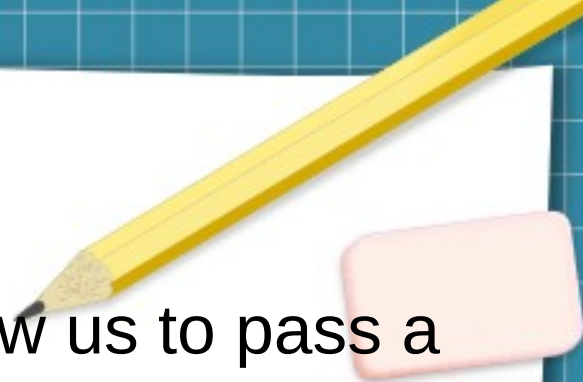
Let's see some *basic searching* and *sorting* algorithms:

- find
- find\_if
- sort
- binary\_search

see [basicSearchSortExamples.cpp](#)

# Function Objects

We saw that *Standard Library algorithms* allow us to pass a **lambda** or a **function pointer**.

A yellow pencil is positioned diagonally in the top right corner, pointing towards the text. Below the pencil's tip is a small, rectangular pink eraser.

# Function Objects

We saw that *Standard Library algorithms* allow us to pass a **lambda** or a **function pointer**.

The **binary\_search** algorithm has an optional *fourth parameter/argument*, a *binary predicate function* with two arguments: search key and element from the collection. The function returns `true` if the search key and the element are equal, and `false` otherwise. This enables the algorithm to search the collections where operator `<` is not provided for the elements of the collection.



# Function Objects



We saw that *Standard Library algorithms* allow us to pass a **lambda** or a **function pointer**.

The **binary\_search** algorithm has an optional *fourth parameter/argument*, a *binary predicate function* with two arguments: search key and element from the collection. The function returns `true` if the search key and the element are equal, and `false` otherwise. This enables the algorithm to search the collections where operator `<` is not provided for the elements of the collection.

Any algorithm that can receive a **lambda** or a **function pointer** can also receive an *object of class that overloads the function-call operator* with a function named **operator()**.

An object of such a class is called a function object and can be used like a **lambda**, a **function**, or a **function pointer**.

# Function Objects



Let's use the accumulate algorithm to find the sum of all the elements in a container using:

- function pointer
- lambda expression
- function object

see [functionObjectExample\\_beginning.cpp](#)

# HW assignment

**(1)** Write a program that allows the user to play with two sets of integer values, with names A and B.

At the beginning of the program the user must be given an opportunity to enter the elements of the sets A and B.

Then he or she should be provided with a menu that will allow to:

- find the intersection of sets A and B
- find the union of sets A and B
- find the difference of sets A and B
- find the difference of sets B and A
- add the elements to the set A
- add the elements to the set B
- empty the set A
- empty the set B
- quit the program

After each iteration, the result must be displayed, and the user should be able to continue playing with the operations.

# HW assignment

(2) Given a vector of decimal values, write the code that uses the algorithm `for_each` and finds the average of all the values in the vector. Here is the beginning of the program:

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

using namespace std;

int main(){
vector<double> v = {1.1, 7.6, 2.3, 9.7, -3.2, -10.4,
7.6, 12.3};
int s; // used to find the sum

for_each(
// put the rest of the code
}
```

**You must use lambda expression or function pointer**

# HW assignment

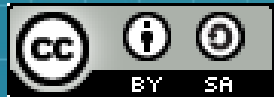
Self-Study:

Section 16.4

Suggested Practice:

Chapter 16 Summary and Self-Review Exercises





This work is licensed under a Creative Commons  
Attribution-ShareAlike 3.0 Unported License.  
It makes use of the works of Mateus Machado Luna.

