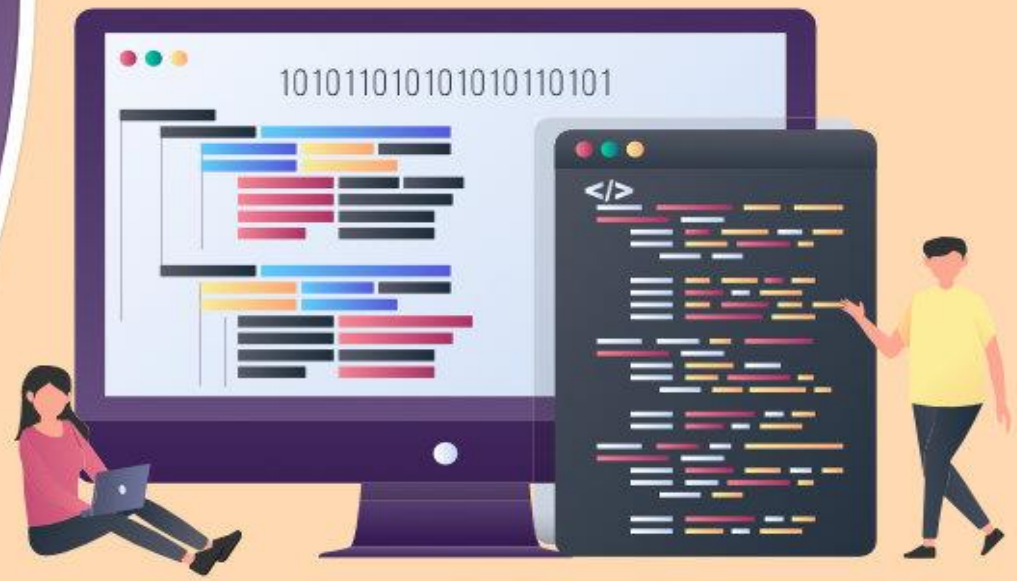
ITERATORS

C++ STL



By – Mohammad Imran

Iterators

Introduction

Iterators are one of the four pillars of the STL. They are very similar to pointers. Like pointers point to the memory address of the variable, iterators point to the memory locations of the elements of the STL containers.

They allow us to access the elements of the container. They reduce the complexity and execution time of the program. Algorithms in STL manipulate data using iterators. The type of container doesn't matter while using iterators, thus making it generic.

By – Mohammad Imran

Iterators

Declaration

Syntax:

```
container_type::iterator iterator_name;
```

Example:

```
vector <int>::iterator itr;
```

Note: Itr is the name of iterator.

By – Mohammad Imran

Iterators

Array

Example - 1

```
#include <iostream>
#include <iterator> //For begin() and end()
using namespace std;
int main()
{
   int arr[] = {1, 2, 3, 4, 5};  // Standard array
   for (auto itr = begin(arr); itr != end(arr); ++itr)
   {
      cout << *itr << " ";
   }
   return 0;
}
```

OUTPUT

```
1 2 3 4 5
```

By – Mohammad Imran

Iterators

The Standard Template Library (STL) provides five types of iterators based on functionality. Each type has specific capabilities and is suited for different tasks. Here's a breakdown of the different types of iterators in STL:

Input Iterator

- ✓ **Functionality:** Can read elements sequentially from a container. It moves only in one direction (forward).

By – Mohammad Imran

Iterators

- ✓ **Capabilities:**
  - Supports only reading (dereferencing) the element.
  - Can increment (++) but cannot move backward.

- ✓ **Examples:** std::istream_iterator, single-pass algorithms (e.g., reading input from a file).
- ✓ **Use case:** Used for reading data from streams or other input sources.

Iterators

Output Iterator

- ✓ **Functionality:** Can write elements to a container sequentially. It also moves only in one direction (forward).
- ✓ **Capabilities:**
  - Supports only writing (dereferencing and assigning values).
  - Can increment (++) but cannot move backward or read from the container.
- ✓ **Examples:** std::ostream_iterator, writing to output streams or containers.
- ✓ **Use case:** Used when writing to containers or streams, such as outputting values to the console.

By – Mohammad Imran

Iterators

Forward Iterator

- ✓ **Functionality:** Can read and write elements sequentially in one direction (forward).
- ✓ **Capabilities:**
  - Can read and write elements (dereference and assign).
  - Can increment (++), but cannot move backward.
  - Allows multiple passes over the container.
- ✓ **Examples:** std::forward_list, std::unordered_set, std::unordered_map.
- ✓ **Use Case:** Suitable for containers that allow only forward traversal, such as singly linked lists.

By – Mohammad Imran

Iterators

Bidirectional Iterator

- ✓ **Functionality:** Can move both forward and backward through a container, supporting reading and writing.
- ✓ **Capabilities:**
  - Can read and write elements (dereference and assign).
  - Can increment (++) and decrement (--) to traverse the container in both directions.
- ✓ **Examples:** std::list, std::set, std::map.
- ✓ **Use Case:** Suitable for containers that allow traversal in both directions, such as doubly linked lists or balanced trees.

Iterators

Random Access Iterator

- ✓ **Functionality:** Provides full flexibility with direct access to any element in constant time. It can move both forward and backward and supports random access to any element.
- ✓ **Capabilities:**
  - Can read and write elements (dereference and assign).Can increment (++), decrement (--), and jump directly to any element using addition or subtraction (+, -).
  - Can compare iterators with relational operators (<, >, <=, >=).

Iterators

- Supports arithmetic operations like itr + n, itr - n, and finding the difference between iterators (itr2 - itr1).

✓ **Examples:** std::vector, std::deque, std::array, std::string.

✓ **Use Case:** Suitable for containers that allow fast, random access to elements like dynamic arrays

Iterators

Iterator Type	Direction	Read	Write	Increment	Decrement	Random Access
Input Iterator	Forward	Yes	No	Yes	No	No
Output Iterator	Forward	Yes	Yes	Yes	No	No
Forward Iterator	Forward	Yes	Yes	Yes	No	No
Bidirectional Iterator	Both	Yes	Yes	Yes	Yes	No
Random Access Iterator	Both	Yes	Yes	Yes	Yes	Yes

By – Mohammad Imran

Algorithm Library in C++

www.educba.com

By – Mohammad Imran

Algorithm

The **<algorithm>** library in C++ Standard Template Library (STL) provides a rich collection of functions to perform various operations on containers such as sorting, searching, manipulating, and more. These algorithms are generic, meaning they work with any container or data structure that supports iterators, making them very flexible and powerful.

By – Mohammad Imran

Algorithm

- ✓ Algorithms in C++ STL define a collection of functions specially designed for use on a sequence of objects.
- ✓ These are standalone template functions, unlike member functions of a container.
- ✓ There are approximately 60 algorithm functions that help save our time and effort.
- ✓ To use algorithms, you must include the **<algorithm>** header file.
- ✓ They also allow you to work with multiple container types simultaneously.

By – Mohammad Imran

Algorithm | Key Feature | Non-Modifying Algorithms

They do not change the contents of the container (e.g., searching, counting).

std::find: Finds the first occurrence of an element.

Example:

```
std::vector<int> v = {1, 2, 3, 4, 5};
auto it = std::find(v.begin(), v.end(), 3);
if (it != v.end()) std::cout << "Element found: " << *it << std::endl;
```

std::count: Counts the number of occurrences of an element.

Example:

```
int count3 = std::count(v.begin(), v.end(), 3);
```

Algorithm | **Key Feature** | **Modifying Algorithms**

They do not change the contents of the container (e.g., searching, counting).

std::find: Finds the first occurrence of an element.

Example:

```
std::vector<int> v = {1, 2, 3, 4, 5};
auto it = std::find(v.begin(), v.end(), 3);
if (it != v.end()) std::cout << "Element found: " << *it << std::endl;
```

By – Mohammad Imran

Non-Modifying Algorithms:

They do not change the contents of the container (e.g., searching, counting).

Modifying Algorithms:

They alter the contents of the container (e.g., sorting, reversing, removing).

Partitioning Algorithms:

Divide a range based on a condition.

Sorting algorithms: Arrange elements in a certain order.

Heap algorithms: Functions for handling elements as heaps.

Coding Questions

By – Mohammad Imran

Question - 1

Write a C++ operational program for file handling according to the given menu.

```
---------MENU----------
1. Create File
2. Write Data
3. Read Data
4. Append Data
5. Delete Data
6. Search Data
7. Update Data
0. Exit
-----------------------
Enter Your Choice :
```

Click here to see code

By – Mohammad Imran

QUIZ

By – Mohammad Imran

Quiz - 1

```
#include <iostream>                    //If file Data.txt not
#include <fstream>                     //found then output is
using namespace std;
int main()
{
   ofstream outfile;
   outfile.open("Data.txt",ios::app);
   if(!outfile)
   {
      cout << "File Not Found" << endl;
      return -1;
   }
   outfile << "Hello World" << endl;
   return 0;
}
```

OUTPUT

Data will inserted in the file

By – Mohammad Imran