C++ STL Tutorial

Hope you have already understood the concept of C++ Template which we have discussed earlier. The C++ STL (Standard Template Library) is a powerful set of C++ template classes to provide general-purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.

At the core of the C++ Standard Template Library are following three well-structured components −

Sr.No	Component & Description
1	**Containers** Containers are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, map etc.
2	**Algorithms** Algorithms act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.
3	**Iterators** Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers.

We will discuss about all the three C++ STL components in next chapter while discussing C++ Standard Library. For now, keep in mind that all the three components have a rich set of pre-defined functions which help us in doing complicated tasks in very easy fashion.

Example

Let us take the following program that demonstrates the vector container (a C++ Standard Template) which is similar to an array with an exception that it automatically handles its own storage requirements in case it grows −

</> Open Compiler

```
#include <iostream>
#include <vector>
using namespace std;

int main() {

   // create a vector to store int
   vector<int> vec;
```

```
   int i;

   // display the original size of vec
   cout << "vector size = " << vec.size() << endl;

   // push 5 values into the vector
   for(i = 0; i < 5; i++) {
      vec.push_back(i);
   }

   // display extended size of vec
   cout << "extended vector size = " << vec.size() << endl;

   // access 5 values from the vector
   for(i = 0; i < 5; i++) {
      cout << "value of vec [" << i << "] = " << vec[i] << endl;
   }

   // use iterator to access the values
   vector<int>::iterator v = vec.begin();
   while( v != vec.end()) {
      cout << "value of v = " << *v << endl;
      v++;
   }

   return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
vector size = 0
extended vector size = 5
value of vec [0] = 0
value of vec [1] = 1
value of vec [2] = 2
value of vec [3] = 3
value of vec [4] = 4
value of v = 0
value of v = 1
value of v = 2
```

```
value of v = 3
value of v = 4
```

Here are following points to be noted related to various functions we used in the above example −

- The push_back() member function inserts value at the end of the vector, expanding its size as needed.
- The size() function displays the size of the vector.
- The function begin() returns an iterator to the start of the vector.
- The function end() returns an iterator to the end of the vector.

C++ Standard Template Library Components

At the core of the C++ Standard Template Library are following four well-structured components −

- Containers
- Algorithm
- Iterators
- Functors (Function Objects)

Containers

Containers are data structures that are used to store and manage collections of data or objects such as vectors, lists, sets, and maps. Here we will see several types of Containers.

1. Sequence Containers

They are containers that store elements in a specific linear order. They allow for direct access to elements and functionalities to manage the order and arrangement of elements.

- **Array** − These are fixed-size collections of elements.
- **Vector** − It's a dynamic array that can resize its size as per requirement.
- **Deque** − Double-ended queue that allows fast insertion and deletion at both ends.
- **List** − A doubly linked list that allows bidirectional iteration.
- **Forward List** − It's a singly linked list that allows efficient insertion and deletion but only traversal in one direction.

- String − In C++ string is also considered a sequential container, it is implemented as a dynamic array of characters that behaves similarly to other sequential containers.

2. Associative Containers

It stores elements in a sorted order which allows fast retrieval based on keys. These containers work in a key-value pair structure, where each element is associated with a unique key. This key is used for quick access to the corresponding value.

- Set − It is a collection of unique elements sorted in a specific order.
- Map − It is a collection of key-value pairs, where keys are unique.
- **Multiset** − It is similar to a set, but allows duplicate elements.
- Multimap − It is similar to a map, but allows duplicate keys.

3. Unordered Associative Containers

It stores elements in an unordered manner allowing for efficient access, insertion, and deletion based on keys. Instead of maintaining a sorted order of elements they use hashing to organize data.

- unordered_set − It is a collection of unique elements, without any specific order.
- unordered_map − It is a collection of key-value pairs without a specific order.
- **unordered_multiset** − It allows duplicate elements without a specific order.
- unordered_multimap − It allows duplicate keys without a specific order.

4. Container Adapters

It provides a different interface for existing containers.

- Stack − It is a data structure that follows the Last In, First Out (LIFO) principle.
- Queue − It is a data structure that follows the First In, First Out (FIFO) principle.
- **Priority Queue** − It is a special type of queue where elements are removed based on priority.

Algorithms

Algorithms in the C++ Standard Template Library (STL) is a big collection of functions which is specifically designed to perform operations on containers. Where these are implemented using iterators which are used to traverse containers without the need to know their internal structure.

Non-modifying Sequence Algorithms

- for_each − It applies in a function to find out each element in a range.
- count − It counts the number of occurrences of a value in a range.
- find() − It finds the first occurrence of a value in a range.

- **find_if** − It finds the first element satisfying a predicate.
- **find_if_not** − It finds the first element not satisfying a predicate.
- **equal** − It checks if two ranges are equal.
- **search** − It searches for a subsequence within a sequence.

1. Modifying Sequence Algorithms

- **copy()** − It copies elements from one range to another.
- **copy_if** − It copies elements satisfying a predicate to another range.
- **copy_n** − It copies a specific number of elements from one range to another.
- **move** − it moves elements from one range to another.
- **transform()** − It applies a function to a range and stores the result.
- **remove** − It removes elements with a specific value from a range.
- **remove_if** − It removes elements satisfying a predicate.
- **unique** − It removes consecutive duplicate elements.
- **reverse()** − It reverses the order of elements in a range.
- **swap()** − It swaps elements.

2. Sorting Algorithms

- **sort** − It sorts elements in a range.
- **stable_sort** − It sorts elements while maintaining the relative order of equivalent elements.
- **partial_sort** − It sorts a portion of a range.
- **nth_element** − It partitions the range such that the nth element is in its final position.

3. Searching Algorithms

- **binary_search** − It checks if an element exists in a sorted range.
- **lower_bound** − It searches for the first position where an element can be inserted to maintain order.
- **upper_bound** − It searches for the position of the first element greater than a specified value.
- **binary_search** − It checks if an element exists in a sorted range.
- **lower_bound** − It finds the first position where an element can be inserted to maintain order.
- **upper_bound** − It searches for the position of the first element greater than a specified value.
- **equal_range** − It returns the range of equal elements.

4. Heap Algorithms

- **make_heap** − It creates a heap from a range.
- **push_heap** − It adds an element to a heap.
- **pop_heap** − It removes the largest element from a heap.
- **sort_heap** − It sorts elements in a heap.

5. Set Algorithms

- **set_union** − It computes the union of two sets.
- **set_intersection** − It computes the intersection of two sets.
- **set_difference** − It computes the difference between two sets.
- **set_symmetric_difference** − It computes the symmetric difference between two sets.

6. Numeric Algorithms

- **accumulate** − it computes the sum (or other operations) of a range.
- **inner_product** − It computes the inner product of two ranges.
- **adjacent_difference** − It computes the differences between adjacent elements.
- **Partial_sum** − it computes the partial sums of a range.

7. Other Algorithms

- **generate** − It fills a range with values generated by a function.
- **shuffle** − It randomly shuffles the elements in a range.
- **partition** − It partitions elements into two groups based on a predicate.

Iterators

Iterators in the C++ Standard Template Library (STL) are objects that act as pointers to the elements within a container which provides a uniform interface for accessing and manipulating data. They serve as a bridge between algorithms and containers.

- **Input Iterators** − They allow read-only access to elements.
- **Output Iterators** − They allow write-only access to elements.
- **Forward Iterators** − They allow reading and writing, which can be incremented.
- **Bidirectional Iterators** − They can be incremented and decremented.
- **Random Access Iterators** − They support arithmetic operations and can access elements directly.

Function Objects (Functors)

A functor (or function object) in C++ is an object that can be called as if it were a function. It can be invoked like regular functions. This capability is achieved by overloading the **function call operator()**.

Here you will explore different types of functors based on the operations they perform.

1. Arithmetic Functors

In C++, arithmetic operators are used to perform basic mathematical operations. Here are the common arithmetic operators along with examples.

- **Addition (+)** − it combines two values to produce their sum.
- **Subtraction (-)** − It calculates the difference between two values.
- **Multiplication (*)** − It multiplies two values to produce their product.
- **Division (/)** − It divides one value by another, resulting in a quotient.
- **Modulus (%)** − It returns the remainder of the division of one value by another.
- **Negate (−)** − It returns the negated value of a parameter.

2. Comparison Functors

Comparison Functors are used for comparing values, especially for sorting or searching in containers.

- **Less Than (<)** − It compares and returns true if the first is less than the second.
- **Greater Than (>)** − It compares and returns true if the first is greater than the second.
- **Less Than or Equal To (≤)** − It compares and returns true if the first is less than or equal to the second.
- **Greater Than or Equal To (≥)** − It compares and returns true if the first is greater than or equal to the second.
- **Equal To (=)** − It compares and returns true if they are equal.
- **Not Equal To (≠)** − It compares and returns true if they are not equal.

3. Logical Functors

Logical functors perform logical operations and can be useful in scenarios involving boolean logic.

- **Logical AND Functor (&&)** − It returns false if at least one or both of the two boolean arguments is false else returns true.
- **Logical OR Functor** (||) − It returns true if at least one of the two boolean arguments or both is true.
- **Logical NOT Functor (!)** − Returns true if the boolean argument is false and vice versa, it returns the opposite to the provided boolean.

4. Bitwise Functors

- **bit_and** − This performs a bitwise AND operation on two operands,
- **bit_or** − This performs a bitwise OR operation on two operands,

- **Bit_xor** − This performs a bitwise exclusive OR (XOR) operation on two operands and returns the output corresponding to it.

Utilities

Utility refers to a collection of general-purpose components which provide basic functionalities in programming, and allow for efficient operations on data and types.

1. Pair and Tuple

- **pair** − It's a container that holds two values of potentially different types, as first and second members.
- **tuple** − A fixed-size collection that can store multiple values of different types, with access to each element via std::get.

2. Smart Pointers

- **unique_ptr** − it's a smart pointer that owns a dynamically allocated object, which automatically deallocates it when the pointer goes out of scope.
- **Shared_ptr** − It's a smart pointer that allows multiple pointers to share ownership of a dynamically allocated object by using reference counting to manage lifetime.
- **Weak_ptr** − It's a smart pointer that provides a non-owning reference to an object managed by std::shared_ptr which prevents circular references.

3. Optional, Variant, and Any Type (C++17)

- **Optional** − It's a wrapper that represents an optional value, indicating whether a value is present or absent.
- **variant** − It's a type-safe union that can hold one of several specified types, providing a way to handle multiple types without ambiguity.
- **any** − It's a type-safe container for storing values of any type, allowing for dynamic type erasure and runtime type checks.

4. Memory Management and Numeric Limits

- **allocator** − This is used for managing dynamic memory.
- **numeric_limits** − It provides information about the properties of fundamental data types.

5. Type Traits

It's a set of templates that allow you to perform compile-time type checks.

- **is_same** − It's a type trait that checks if two types are the same, results true if they are else false.

- **is_integral** – It's a type trait that determines if a given type is an integral type (e.g., int, char, long), returning true for integral types and false for others.
- **is_floating_point** – It's a type trait that checks if a given type is a floating-point type (e.g., float, double, long double), returning true for floating-point types and false for non-floating-point types.
- **enable_if** – It's a utility that enables or disables template instantiation based on a compile-time condition which is often used for SFINAE (Substitution Failure Is Not An Error).

Basic STL Features: Simple Example

Here's a simple example demonstrating some basic STL features

</> Open Compiler

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

int main() {
   std::vector<int> vec = {5, 2, 8, 1, 3};

   // Sorting the vector
   std::sort(vec.begin(), vec.end());

   // Displaying sorted elements
   std::cout << "Sorted vector: ";
   for (int num : vec) {
      std::cout << num << " ";
   }
   std::cout << std::endl;

   // Finding an element
   auto it = std::find(vec.begin(), vec.end(), 3);
   if (it != vec.end()) {
      std::cout << "Element 3 found at position: "
         << std::distance(vec.begin(), it) << std::endl;
   } else {
      std::cout << "Element 3 not found" << std::endl;
   }

   return 0;
}
```

Output

```
Sorted vector: 1 2 3 5 8
Element 3 found at position: 2
```

Explanation

This code provides the use of STL in various places,

- **std::vector<int> vec** − This above program used std::vector, which is an STL container that allows dynamic array management.
- **std::sort(vec.begin(), vec.end())** − This function is a part of STL algorithms which operate on containers, which sorts the elements of the vector in ascending order.
- **std::find(vec.begin(), vec.end(), 3)** − This function searches for the value 3 in the vector and returns an iterator to it if found.
- **begin() and vec.end()** − These functions return iterators that point to the beginning and just past the end of the vector, respectively.
- **std::distance(vec.begin(), it)** − This function calculates the number of elements between two iterators (in this case, from the beginning of the vector to the found iterator), effectively giving the index of the found element.

Benefits of Using STL

Using the Standard Template Library (STL) in C++ has several benefits because it provides a big collection of pre-defined data structures and algorithms, which saves time and effort and reduces the need to implement these from scratch. It also promotes code reuse, modularity, type safety, and flexibility. Which makes the same code work with different data types. Overall it enhances programming efficiency and code quality.