Templates

Template provides the idea of generic classes. Use of one function or class that works for all data types is generalization. With the help of templates and functions we can create generic data types and idea leads to generic programming. In the generic programming generic data types are passed as argument to function and classes.

- Template provides the idea of generic classes.
- Use of one function or class that works for all data types is generalization.
- With the help of templates and functions we can create generic data types and this idea leads to generic programming.
- In the generic programming generic data types are passed as argument to function and classes.
- A template is similar to a macro which can work for different types of data.
- A template created for a function so a function works for variety of data types is termed as function template.
- For example a function template max is written which finds maximum of two integer, two floats, two chars etc. Similarly, when a template is written for a class so one single class works for variety of data types is termed as class template.

A template may be considered as a kind of macro. When the actual object of that type is to be defined, the template definition is substituted with required data type. For example, if we define a template Array of elements, then this same generic definition may be used to create Array of integers or of characters or float quantities.

Function Template

- Function templates can be used to create a family of functions that may take different arguments.
- A function template does not occupy space in memory.
- The actual definition of function template is generated when function is called with specific data type.
- The function template does not result in saving memory.
- A function template can be defined as follows:

```
template <class  T>                    or       template <typename  T>
return_type function_name (arguments of type T)
{
...........
..........body of function with argument of type T
...........
};
```

Before function definition we write template which is a keyword. In the brackets < and > we write class and any name which serves as the generic type. The name may be a single character or a word (similar to identifier). The name usually written in capital but may be in small case too.

The following example demonstrates creation of a function template swap:

```
template <class T>
void swap(T & x, T& y)
{
T temp = x;
x = y;
y = temp:
};
```

The swap() function can now be invoked like any ordinary function. Any call to swap() with input arguments will exchange the values contained in those arguments. Hence if a and b are integer variables and p and q are float variables; we may invoke swap() function as swap(a,b) and swap(p,q), respectively. The same function definition can be used to swap values of two variables of different types.

```
#include <iostream>
template<class T>
void bsort(T a[], int n)
{
for (int i=0; i<n-1; i++)
for (int j=n-1; i<j; j--)
if (a[j] < a[j-1])
swap(a[j], a[j-1]);
}
template <class X>
void swap( X &a, X &b)
{
X temp =a;
a = b;
b = temp;
}
int main()
{
int x[5] = {10,50,30,60,40};
float y[5] = {3.2, 71.5, 17.3, 45.9, 92.7};
bsort(x,5);
bsort(y,5);
cout << “Sorted X-Array:”;
for (int i=0; i<5; i++)
cout << x[i] << “ ”;
cout << endl;
cout << “Sorted Y-Array:”;
for (int j=0; j<5; j++)
cout << y[j] << “ ”;
cout << endl;
return(0);
};
```

This program uses two function templates swap() and bsort(). The function template swap() is invoked within the bsort() function and is hence said to be nested in it. This program can be used to sort different types of lists without the need of modifying the program.

The program will produce following output:

```
Sorted X-Array: 10 30 40 50 60
```

Write a program to find maximum of two number using Function Template

```
#include<iostream>
using namespace std;
template <typename T>
T getmax(T x, T y)
{
        return x>y?x:y;
}
int main()
{
        int x=18, y=20;
cout<<getmax<int>(x,y)<<endl;
}
```

- Write a program to sort array elements using Function Template
- Write a program using function template to find maximum value stored in the array.
- Working with two generic generic data type at a time
- Checking equality of data type of two variables using Function Template
- Checking equality of three data types using Function Template
- Mixing built-in type with generic type
- Overloading Function Template

Class Templates

We need not make a new class definition every time. We define a generic class with a parameter that s replaced by a particular data type at the time of actual use of that class. This is the reason template classes are also known as parameterized classes.

The general syntax for defining a class template is:

```
template <class T>
class classname
{
...........
..........class specification with anonymous type T
...........
};
```

Example:

```
template<class T>
class vector
{
T * v; // the vector is of type T
int size;
Public:
vector(int m)
{
v = new T [size = m];
for (int i=0; i<size; i++)
v[i]=0;
}
vector (T * a)
{
for(int i=0; i<size; i++)
v[i] = a[i];
}
T operator * (vector &x)
{
T sum = 0;
for(int i=0; i<size; i++)
sum += this->v[i] *x- v[i];
return (sum);
}
};
```

This definition creates a template class vector of type T with variables, constructors and „*" operator. This class definition is similar to an ordinary class definition except that of the use of template<class T> and use of type T inside the class definition. The template<class T> tells the compiler that it is a template class with parameter T instead of a normal class definition. The declaration thus creates a parameterized class with T as the parameter, which can be substituted with any valid data type. This can be done by a statement of the following form:

```
classname <type> objectname(argument list);
```

Here we have written only one class definition for class vector but we have been able to create more than one actual instantiations of the template class vector. A class template can also be created by using multiple generic data types as arguments. The general syntax for such definition would be as below:

```
template <class T1, class T2, .......>
class classname
{
...........
..........class specification with anonymous type T
...........
};
```

```
#include <iostream>

template<class T1, class T2>
class Example
{
T1 x;
T2 y;
Public:
Example(T1 a, T2 b)
{
x = a;
y = b;
}
void show ()
{
cout << x << “and” << y << “\\n”;
}
};

int main()
{
Example <float, int> test1 (3.45, 345);
Example <int, char> test2 (100, „m‟);
test1.show();
test2.show();
return(0);
};
```

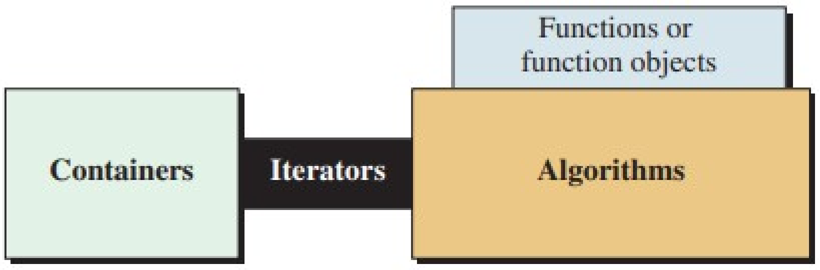
The program creates two template classes test1 and test2 using the template class Example. The test1 class has two parameter values “3.45” and “345”, whereas test2 class has two parameter values “100” and character “m”. For creating test1 object, arguments are float and integer respectively, whereas in case of test2 object they are integer and character. The values displayed in invocation of show() function from main will be “3.45 and 345” for test1 and “100 and m” for test2 object.

Standard Template Library (STL)

The Standard Template Library (STL) is a set of template classes that provides many basic algorithms or containers to code quickly, efficiently and in generic way.

• It must be convenient, efficient, useful, and safe for all users.
• Users should not have to reprogram the library in order to find it useful.
• The library must provide a simple interface.
• The library must be complete in what it tries to do.
• The library must blend well with built-in data types and keywords.

The standard library is organized into a set of header files, each serving a different purpose. STL is comprised of four components.



Containers can be described as the objects that hold the data of the same type. Containers are used to implement different data structures. Containers are the heart of the STL. The library defines four categories of containers, each for a different purpose: sequence containers, associate containers, container adapters, and pseudo-containers. Each container category has been defined for a group of applications.

Algorithms are operations that we apply to the container elements. They are divided into four categories: non-mutating algorithms, which do not change the container structure; mutating algorithms, which change the structure; sorting algorithms, which reorder elements in a container; and numeric algorithms, which apply mathematical operations to the elements.

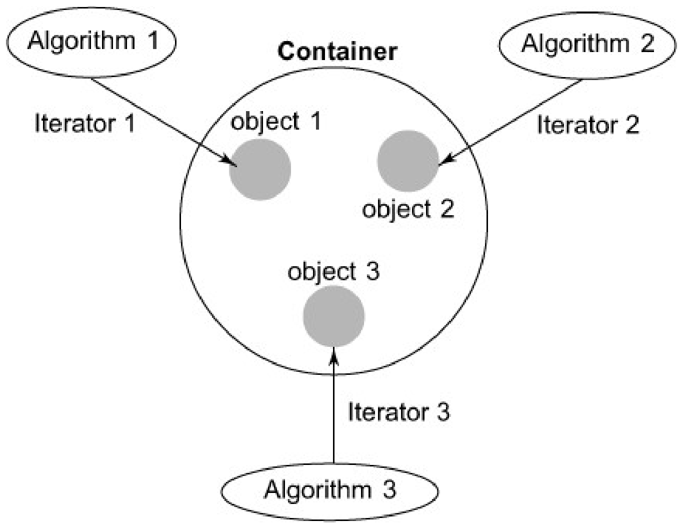
Iterators An iterator allows us to access each element individually and apply the desired operation to it. This means that we do not need one algorithm that counts one type and another that counts another type. An algorithm can be applied to any container that provides the type of iterators that the container supports.

Functions and Function Objects To apply algorithms to the container, STL uses functions or function objects in the algorithm definition. They allow the STL to define a generic algorithm and use functions or function objects to make the algorithm specific. In the first case, we need a function definition; in the second case, we need a class for which the operator() is defined. The user defines a function for the first case, but the class is normally defined in the STL library and the user can only call the constructor of the class.

STL Components

The STL contains several components. But at its core are three key components.
They are:

1. Iterators
2. Containers
3. Algorithms



Algorithms employ iterators to perform operations stored in containers.

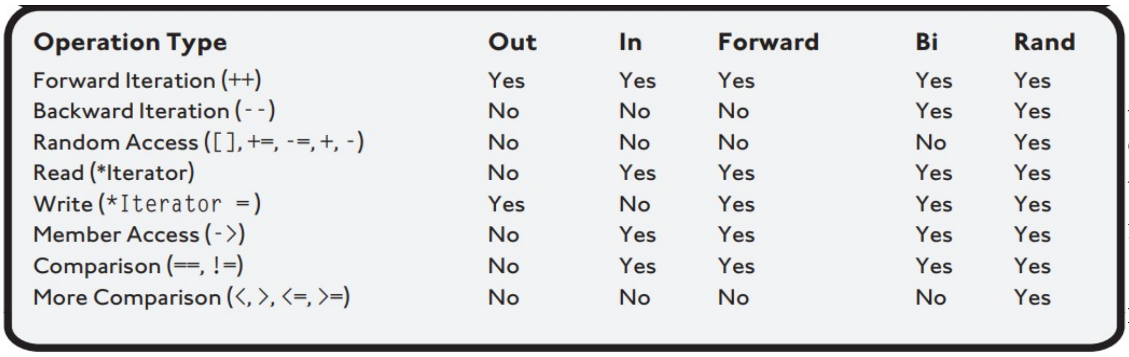
A ***container*** is an object that actually stores data. It is a way data is organized in memory. The STL containers are implemented by template classes and therefore can be easily customized to hold different types of data.

An ***iterator*** is an object (like a pointer) that points to an element in a container. We can use iterators to move through the contents of containers. Iterators are handled just like pointers. We can increment or decrement them. Iterators connect algorithms with containers and play a key role in the manipulation of data stored in the containers.

1. ITERATORS

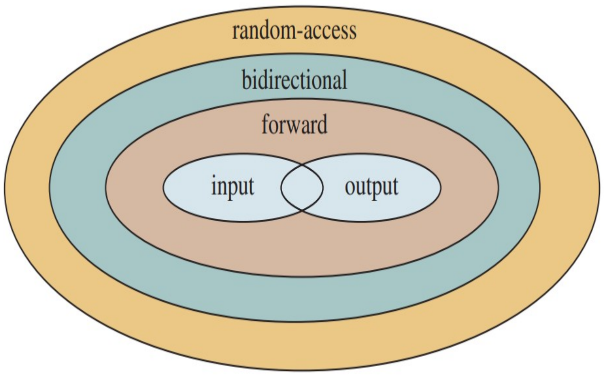
Iterators are those entities that helps in traversing through elements. It is an object (like a pointer) that points to an element (similar to a pointer pointing to some location) inside the container. It can traversed from one element to the next element in the range by incrementing. The most obvious form of an iterator is a pointer. A pointer can point to elements in an array and can iterate through them using the increment operator (++). But, all iterators do not have similar functionality as that of pointers. Depending upon the functionality of iterators they can be classified into five categories, with the outer one being the most powerful one and consequently the inner one is the least powerful in terms of functionality. They are input, output, forward, bi-directional, and random.

An **iterator** allows us to access each element individually and apply the desired operation to it. An **algorithm** can be applied to any container that provides the type of iterators that the container supports.



Operation Type	Out	In	Forward	Bi	Rand
Forward Iteration (++)	Yes	Yes	Yes	Yes	Yes
Backward Iteration (--)	No	No	No	Yes	Yes
Random Access ([ ], +=, -=, +, -)	No	No	No	No	Yes
Read (*Iterator)	No	Yes	Yes	Yes	Yes
Write (*Iterator =)	Yes	No	Yes	Yes	Yes
Member Access (->)	No	Yes	Yes	Yes	Yes
Comparison (==, !=)	No	Yes	Yes	Yes	Yes
More Comparison (<, >, <=, >=)	No	No	No	No	Yes

An iterator contains two functions:
begin(): It returns iterator to the first element



end(): It returns the iterator to the last element.

1.1 Types of Iterator

1.1.1 Input iterator:

- An Input iterator is an iterator that allows the program to read the values from the container.
- Dereferencing the input iterator allows us to read a value from the container, but it does not alter the value.
- An Input iterator is a one way iterator.
- An Input iterator can be incremented, but it cannot be decremented.

Equality / Inequality Comparison

A == B // Checking for equality
A != B // Checking for inequality

Dereferencing

***A   // Dereferencing using ***
A -> m   // Accessing a member element m

Increment operation

A++   // Using post increment operator

++A   // Using pre increment operator

Cannot be decremented: Just like we can use operator ++() with input iterators for incrementing them, we cannot decrement them.

If A is an input iterator, then

```
A--    // Not allowed with input iterators
```

Use in multi-pass algorithms: Since it is unidirectional and can only move forward, therefore, such iterators cannot be used in multi-pass algorithms, in which we need to process the container multiple times.

Relational Operators: Although, input iterators can be used with equality operator (==), but it can not be used with other relational operators like <=.

If A and B are input iterators, then

```
A == B     // Allowed
A <= B     // Not Allowed
```

Arithmetic Operators: Similar to relational operators, they also can't be used with arithmetic operators like +, – and so on. This means that input operators can only move in one direction that too forward and that too sequentially.

If A and B are input iterators, then

```
A + 1     // Not allowed
B - 2     // Not allowed
```

1.1.2 Output iterator:

- An output iterator is similar to the input iterator, except that it allows the program to modify a value of the container, but it does not allow to read it.
- It is a one-way iterator.
- It is a write only iterator.

Equality / Inequality Comparison: Unlike input iterators, output iterators cannot be compared for equality with another iterator. So, the following two expressions are invalid if A and B are output iterators:

```
A == B  // Invalid - Checking for equality
A != B  // Invalid - Checking for inequality
```

Dereferencing: An input iterator can be dereferenced as an rvalue, using operator * and ->, whereas an **output iterator can be dereferenced as an lvalue** to provide the location to store the value.

So, the following two expressions are valid if A is an output iterator:

```
*A = 1      // Dereferencing using *
A -> m = 7   // Assigning a member element m
```

Incrementable: An output iterator can be incremented, so that it refers to the next element in sequence, using operator ++().

So, the following two expressions are valid if A is an output iterator:

```
A++   // Using post increment operator
++A   // Using pre increment operator
```

Swappable: The value pointed to by these iterators can be exchanged or swapped.

Cannot be decremented: Just like we can use operator ++() with output iterators for incrementing them, we cannot decrement them.

If A is an output iterator,then

```
A--    // Not allowed with output iterators
```

Use in multi-pass algorithms: Since, it is unidirectional and can only move forward, therefore, such iterators cannot be used in multi-pass algorithms, in which we need to move through the container multiple times.

Relational Operators: Just like output iterators cannot be used with equality operators (==), it also can not be used with other relational operators like =.

If A and B are output iterators, then

```
A == B     // Not Allowed
A <= B     // Not Allowed
```

Arithmetic Operators: Similar to relational operators, they also can't be used with arithmetic operators like +, – and so on. This means that output operators can only move in one direction that too forward and that too sequentially.

If A and B are output iterators, then

```
A + 1     // Not allowed
B - 2     // Not allowed
```

1.1.3 Forward iterator:

- Forward iterator is a **combination of input as well as output iterators,** uses the ++ operator to navigate through the container.
- Can be used in multi-pass algorithms.
- Forward iterator goes through each element of a container and one element at a time.

Equality / Inequality Comparison: A forward iterator can be compared for equality with another iterator. Since, iterators point to some location, so the two iterators will be equal only when they point to the same position, otherwise not.

So, the following two expressions are valid if A and B are forward iterators:

```
A == B  // Checking for equality
A != B  // Checking for inequality
```

Dereferencing: Because an input iterator can be dereferenced, using the operator * and -> as an rvalue and an output iterator can be dereferenced as an lvalue, so forward iterators can be used for both the purposes.

Incrementable: A forward iterator can be incremented, so that it refers to the next element in sequence, using operator ++().

Note: The fact that we can use forward iterators with increment operator doesn't mean that operator --() can also be used with them. Remember, that forward iterators are unidirectional and can only move in the forward direction.

So, the following two expressions are valid if A is a forward iterator:

```
A++   // Using post increment operator
++A   // Using pre increment operator
```

Swappable: The value pointed to by these iterators can be exchanged or swapped.

Can not be decremented: Just like we can use operator ++() with forward iterators for incrementing them, we cannot decrement them. Although it is higher in the hierarchy than input and output iterators, still it can't overcome this deficiency.

That is why, its name is forward, which shows that **it can move only in forward direction**.

If A is a forward iterator, then

```
A--    // Not allowed with forward iterators
```

Relational Operators: Although, forward iterators can be used with equality operator (==), it can not be used with other relational operators like =.

If A and B are forward iterators, then

```
A == B     // Allowed
A <= B     // Not Allowed
```

Arithmetic Operators: Similar to relational operators, they also can't be used with arithmetic operators like +, – and so on. This means that forward operators can only move in one direction that too forward and that too sequentially.

If A and B are forward iterators, then

A + 1   // Not allowed
B - 2   // Not allowed

Use of offset dereference operator ([ ]): Forward iterators do not support offset dereference operator ([]), which is used for random-access.

If A is a forward iterator, then
A[3]   // Not allowed

1.1.4 Bidirectional iterator:

- A Bidirectional iterator is similar to the forward iterator, except that it also moves in the backward direction.
- It is a two way iterator.
- It can be incremented as well as decremented.

Equality / Inequality Comparison: A Bidirectional iterator can be compared for equality with another iterator. Since, iterators point to some location, so the two iterators will be equal only when they point to the same position, otherwise not.

So, the following two expressions are valid if A and B are Bidirectional iterators:

A == B // Checking for equality
A != B // Checking for inequality

Dereferencing: Because an input iterator can be dereferenced, using operator * and -> as an rvalue and an output iterator can be dereferenced as an lvalue, so forward iterators being the combination of both can be used for both the purposes, and similarly, bidirectional operators can also serve both the purposes.

Incrementable: A Bidirectional iterator can be incremented, so that it refers to the next element in sequence, using operator ++().

So, the following two expressions are valid if A is a bidirectional iterator:

A++   // Using post increment operator
++A   // Using pre increment operator

Decrementable: This is the feature which differentiates a Bidirectional iterator from a forward iterator. Just like we can use operator ++() with bidirectional iterators for incrementing them, we can also decrement them.

That is why, its name is bidirectional, which shows that **it can move in both directions**.

Swappable: The value pointed to by these iterators can be exchanged or swapped

Relational Operators: Although, Bidirectional iterators can be used with equality operator (==), but it can not be used with other relational operators like , =.

If A and B are Bidirectional iterators, then

A == B   // Allowed
A <= B   // Not Allowed

Arithmetic Operators: Similar to relational operators, they also can’t be used with arithmetic operators like +, – and so on. This means that bidirectional iterators can move in both the direction, but sequentially.

If A and B are Bidirectional iterators, then

```
A + 1     // Not allowed
B - 2     // Not allowed
```

Use of offset dereference operator ([ ]): Bidirectional iterators doesnot support offset dereference operator ([ ]), which is used for random-access.

If A is a Bidirectional iterator, then

```
A[3]    // Not allowed
```

1.1.5 Random Access Iterator:

- Random access iterator can be used to access the **random element of a container.**
- Random access iterator has all the features of a bidirectional iterator, and it also has one more additional feature, i.e., pointer addition. By using the pointer addition operation, we can access the random element of a container.
- Most complete iterators in terms of functionality

Usability: Random-access iterators can be used in multi-pass algorithms, i.e., an algorithm which involves processing the container several times in various passes.

Equality/Inequality Comparison: A Random-access iterator can be compared for equality with another iterator. Since, iterators point to some location, so the two iterators will be equal only when they point to the same position, otherwise not.

So, the following two expressions are valid if A and B are Random-access iterators:

```
A == B  // Checking for equality
A != B  // Checking for inequality
```

Dereferencing: A random-access iterator can be **dereferenced both as an rvalue as well as an lvalue.**

Incrementable: A Random-access iterator can be incremented, so that it refers to the next element in the sequence, using operator ++(), as seen in the previous code, where i1 was incremented in the for loop.

So, the following two expressions are valid if A is a random-access iterator:

```
A++   // Using post increment operator
++A   // Using pre increment operator
```

Decrementable: Just like we can use operator ++() with Random-access iterators for incrementing them, we can also decrement them.

Relational Operators: Although, Bidirectional iterators cannot be used with relational operators like =, random-access iterators being higher in hierarchy support all these relational operators.

If A and B are Random-access iterators, then

```
A == B     // Allowed
A <= B     // Allowed
```

Arithmetic Operators: Similar to relational operators, they also can be used with arithmetic operators like +, – and so on. This means that Random-access iterators can move in both the direction and that too randomly.

If A and B are Random-access iterators, then

```
A + 1     // Allowed
B - 2     // Allowed
```

Use of offset dereference operator ([ ]): Random-access iterators support offset dereference operator ([ ]), which is used for random-access.

If A is a Random-access iterator, then
A[3]   // Allowed

Program: Write the definition of a generic function that prints the value of all elements in a vector of any type.

1.2 Iterator Functions

Begin() Returns iterator to the beginning of the container

End() Returns an iterator to the element following the last element of the container This element acts as a placeholder, attempting to access it results in undefined behavior.

Rbegin() (reverse iterator) Returns iterator to the last element of the container

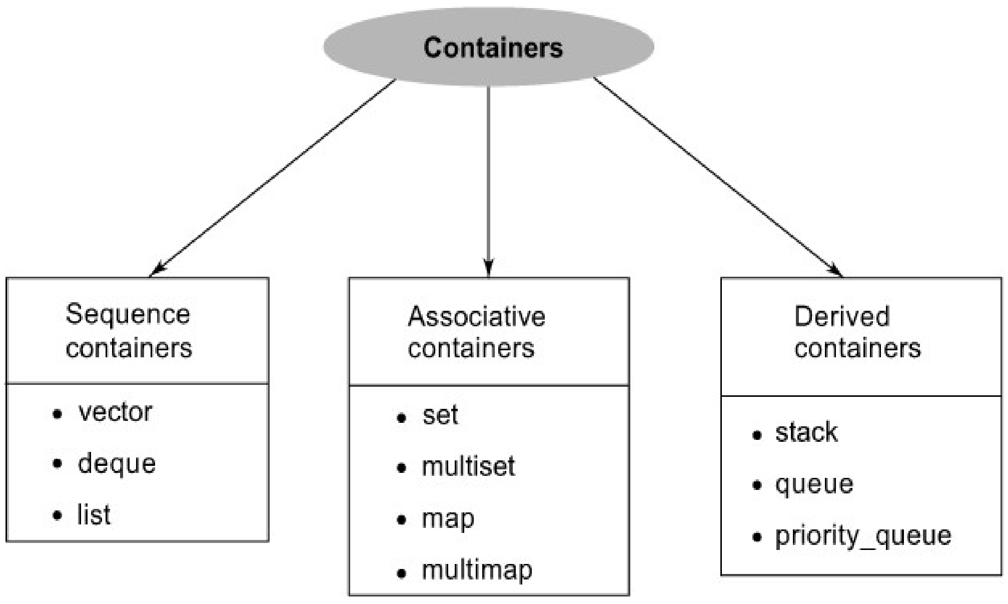
Rend() Returns a reverse iterator pointing to the theoretical element right before the first element in the array container

1.3 Iterator Invalidation

Due to update (Insert/ Delete) of container that is using iterators. No run time error, but iterator no longer guaranteed to have access to the same element after update. Well-documented rules depends on container implementation.

2. CONTAINERS

Containers are objects that hold data (of same type). The STL defines ten containers which are grouped into three categories. Each container class defines a set of functions that can be used to manipulate its contents. For example, a vector container defines functions for inserting elements, erasing the contents, and swapping the contents of two vectors.

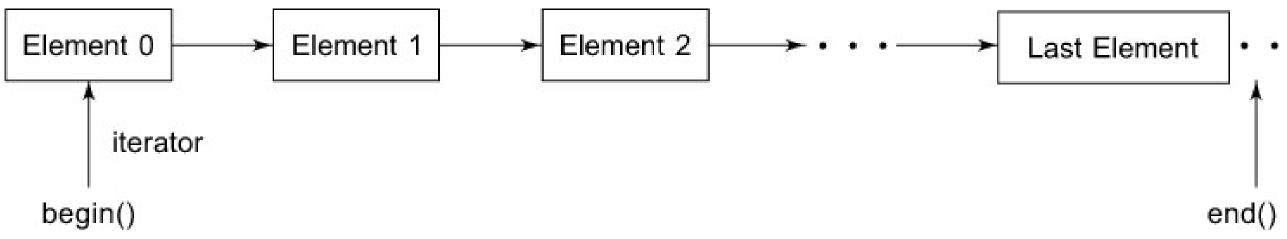


Containers supported by the STL

Container	*Description*	*Header file*	*Iterator*
vector	A dynamic array. Allows insertions and deletions at back. Permits direct access to any element	<vector>	Random access
list	A bidirectional, linear list. Allows insertions and deletions anywhere.	<list>	Bidirectional
deque	A double-ended queue. Allows insertions and deletions at both the ends. Permits direct access to any element.	<deque>	Random access
set	An associate container for storing unique sets. Allows rapid lookup. (No duplicates allowed)	<set>	Bidirectional
multiset	An associate container for storing non-unique sets. (Duplicates allowed)	<set>	Bidirectional
map	An associate container for storing unique key/value pairs. Each key is associated with only one value (One-to-one mapping). Allows key-based lookup.	<map>	Bidirectional
multimap	An associate container for storing key/value pairs in which one key may be associated with more than one value (one-to-many mapping). Allows key-based lookup.	<map>	Bidirectional
stack	A standard stack. Last-in-first-out(LIFO)	<stack>	No iterator
queue	A standard queue. First-in-first-out(FIFO).	<queue>	No iterator
priority– queue	A priority queue. The first element out is always the highest priority element.	<queue>	No iterator

2.1 Sequence Containers

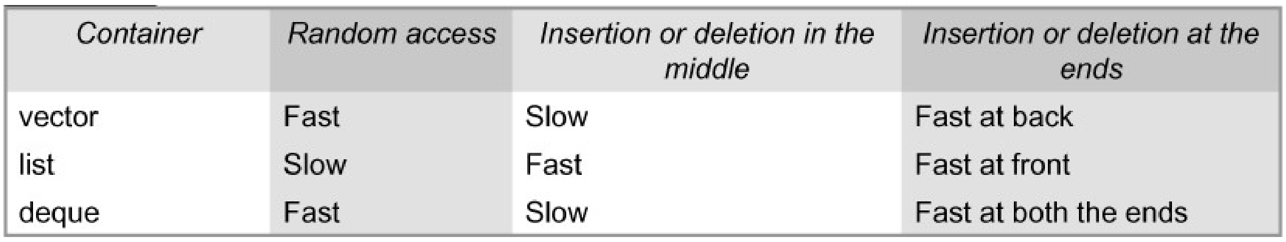
Sequence containers store elements in a linear sequence, like a line. Each element is related to other elements by its position along the line. They all expand themselves to allow insertion of elements and all of them support a number of operations on them.



The STL provides three types of sequence containers:

1. **Vector**
2. **List**
3. **Deque**

Elements in all these containers can be accessed using an **iterator**. The difference between the three of them is related to only their performance. Comparison performance in terms of speed of random access and insertion or deletion of elements is shown below.



Container	*Random access*	*Insertion or deletion in the middle*	*Insertion or deletion at the ends*
vector	Fast	Slow	Fast at back
list	Slow	Fast	Fast at front
deque	Fast	Slow	Fast at both the ends

Comparison of sequence containers

2.1.1 Vectors

The vector is the most widely used container. It stores elements in contiguous memory locations and enables direct access to any element using the subscript operator []. Vectors are similar to arrays. A vector can change its size dynamically and therefore allocates memory as needed at run time. The vector container supports random access iterators, and a wide range of iterator operations may be applied to a vector iterator. Class vector supports a number of constructors for creating vector objects.

Function	Task
at( )	Gives a reference to an element
back( )	Gives a reference to the last element
begin( )	Gives a reference to the first element
capacity( )	Gives the current capacity of the vector
clear( )	Deletes all the elements from the vector
empty( )	Determines if the vector is empty or not
end( )	Gives a reference to the end of the vector
erase( )	Deletes specified elements
insert( )	Inserts elements in the vector
pop_back( )	Deletes the last element
push_back( )	Adds an element to the end
resize( )	Modifies the size of the vector to the specified value
size( )	Gives the number of elements
max_size()	Gives the maximum size of vector
swap( )	Exchanges elements in the specified two vecto

Program:-
Write a program that accepts a shopping list of five items from the keyboard and stores them in a vector. Extend the program to accomplish the following:

(a) To delete a specified item in the list.
(b) To add an item at a specified location.
(c) To add an item at the end.
(d) To print the contents of the vector.

Lists

- It supports a bidirectional, linear list and provides an efficient implementation for deletion and insertion operations.
- Rapid insertion and deletion
- It can be accessed sequentially only, not support random access
- Bidirectional iterators are used for accessing list elements.
- Any algorithm that requires input, output, forward, or bidirectional iterators can operate on a list.
- Class list provides many member functions for manipulating the elements of a list.
- Header file **<list>** must be included to use the container class list.
- No operator[]
- Best suited for large structure

Function	**Task**
back( )	Gives reference to the last element
begin( )	Gives reference to the first element
clear( )	Deletes all the elements
empty( )	Decides if the list is empty or not
end( )	Gives reference to the end of the list
erase( )	Deletes elements as specified
front()	Gives reference to the first element
insert( )	Inserts elements as specified
merge( )	Merges two ordered lists
pop_back()	Deletes the last element
pop_front()	Deletes the first element
push_back()	Adds an element to the end
push_front( )	Adds an element to the front
remove( )	Removes elements as specified
resize( )	Modifies the size of the list
reverse( )	Reverses the list
size( )	Gives the size of the list
sort( )	Sorts the list
splice( )	Inserts a list into the invoking list
swap( )	Exchanges the elements of a list with those in the invoking list
unique( )	Deletes the duplicating elements in the list

Important member functions of the list class

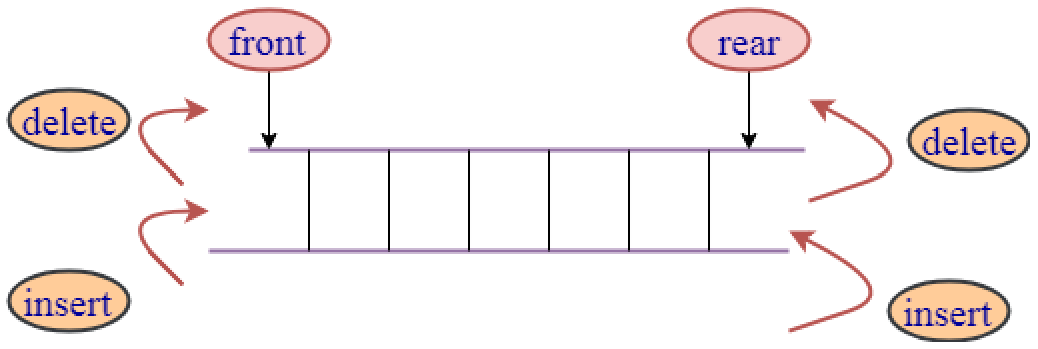
Deque

The name deque (pronounced deck) is short for double-ended queue. This means that we can insert and erase at both ends of a deque.

However, the extra capability to insert at the beginning makes a deque less efficient than a vector because doing so requires that the system allocate extra memory to extend the deque at either end. This means if we need a structure that does not need insertion and erasure in the front, it is more efficient to use a vector than a deque. Insertion and erasure in the middle of a deque have the same inefficiency as in the case of a vector.

Syntax:

deque< object_type > deque_name;



Method	Description
assign()	It assigns new content and replacing the old one.
emplace()	It adds a new element at a specified position.
emplace_back()	It adds a new element at the end.
emplace_front()	It adds a new element in the beginning of a deque.
insert()	It adds a new element just before the specified position.
push_back()	It adds a new element at the end of the container.
push_front()	It adds a new element at the beginning of the container.
pop_back()	It deletes the last element from the deque.

pop_front()	It deletes the first element from the deque.
swap()	It exchanges the contents of two deques.
clear()	It removes all the contents of the deque.
empty()	It checks whether the container is empty or not.
erase()	It removes the elements.
max_size()	It determines the maximum size of the deque.
resize()	It changes the size of the deque.
shrink_to_fit()	It reduces the memory to fit the size of the deque.
size()	It returns the number of elements.
at()	It access the element at position pos.
operator[]()	It access the element at position pos.
operator=()	It assigns new contents to the container.
back()	It access the last element.
begin()	It returns an iterator to the beginning of the deque.
cbegin()	It returns a constant iterator to the beginning of the deque.
end()	It returns an iterator to the end.
cend()	It returns a constant iterator to the end.
rbegin()	It returns a reverse iterator to the beginning.
crbegin()	It returns a constant reverse iterator to the beginning.
rend()	It returns a reverse iterator to the end.

crend()	It returns a constant reverse iterator to the end.
front()	It access the last element.

2.2 Associative Containers

Associative containers are designed to support direct access to
elements using keys. They are not sequential. There are four types of
associative containers:

1. Map
2. Multimap
3. Set
4. Multiset

All these containers store data in a structure called **tree** which facilitates ***fast searching, deletion, and insertion***. However, these are very slow for random access and inefficient for sorting.

Elements in an associative container are stored and retrieved by a key. To access an element in an associate container, we use the key of the element. Associative containers are normally implemented as a binary search tree.

Containers **map** and **multimap** are used to store pairs of items, one called the ***key*** and the other called the ***value***. We can manipulate the values using the keys associated with them. The values are sometimes called **mapped values.**

The main difference between a map and a multimap is that a map allows only **one key** for a given value to be stored while multimap permits **multiple keys**.

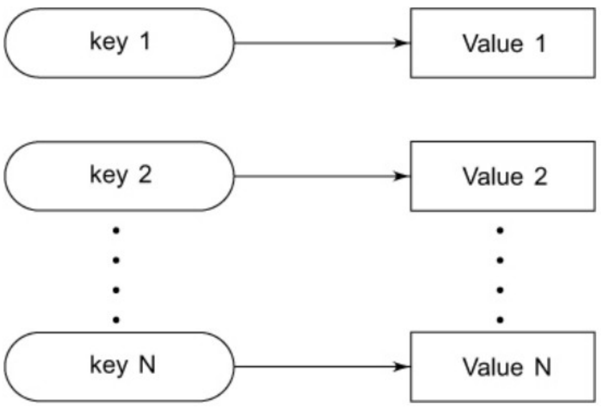
Containers **set** and **multiset** can store a number of items and provide operations for manipulating them using the values as the keys.

For example, a **set** might store objects of the **student** class which are ordered alphabetically using names as keys. We can search for a desired student using his name as the key. The main difference between a **set** and a **multiset** is that a **multiset** allows duplicate items while a **set** does not.

Application: You have a large collection of objects that you would like to be ordered and frequently have to look up to find specific information. set and map containers that can automatically sort their objects and provide very fast search characteristics.

2.2.1 Maps

A map is a sequence of (key, value) pairs where a single value is associated with each unique key. We should specify the key to obtain the associated value.



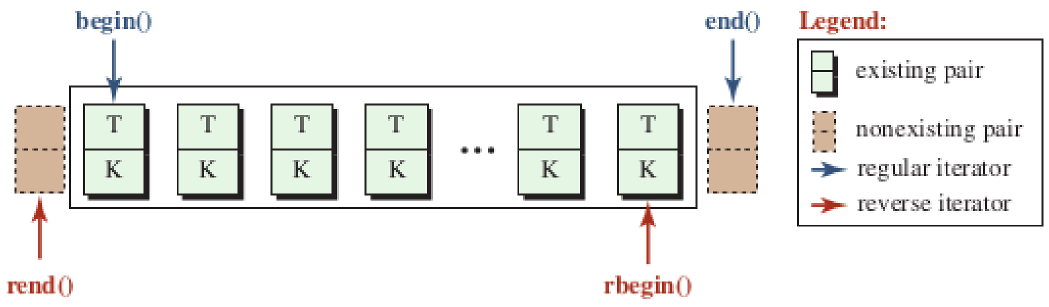
A map is commonly called an associative array. The key is specified using the subscript operator [] as shown below:

```
phone["John"] = 1111;
```

This creates an entry for "John" and associates (i.e., assigns) the value 1111 to it. phone is a map object. We can change the value, if necessary, as follows:

```
phone["John"] = 9999;
```

This changes the value 1111 to 9999. We can also insert and delete pairs anywhere in the map using insert( ) and erase( ) functions. Important member functions of the map class are listed in



Notes:
The keys are unique in a map.
The objects are in nonlinear relationship to each other.
The *map* class uses bidirectional iterators.

Function	Task
begin( )	Gives reference to the first element
clear( )	Deletes all elements from the map
empty( )	Decides whether the map is empty or not
end( )	Gives a reference to the end of the map
erase( )	Deletes the specified elements
find( )	Gives the location of the specified element
insert( )	Inserts elements as specified
size( )	Gives the size of the map
swap( )	Exchanges the elements of the given map with those of the invoking map
operator[]	To access the elements in the map with the given key value.

Searching in std::map

- To get the iterator of the first occurrence of a key, the find() function can be used. It returns end() if the key does not exist.

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
auto it = mmp.find(6);
if(it!=mmp.end())
        std::cout << it->first << ", " << it->second << std::endl;    //prints: 6, 5
else
        std::cout << "Value does not exist!" << std::endl;
```

- Another way to find whether an entry exists in std::map or in std::multimap is using the count() function, which counts how many values are associated with a given key. Since std::map associates only one value with each key, its count() function can only return 0 (if the key is not present) or 1 (if it is). For std::multimap , count() can return values greater than 1 since there can be several values associated with the same key.

```
std::map< int , int > mp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };

if(mp.count(3) > 0)            // 3 exists as a key in map

        std::cout << "The key exists!" << std::endl;
        // This line would be executed.

else
        std::cout << "The key does not exist!" << std::endl;
```

In the case of std::multimap, there could be several elements having the same key. To get this range, the equal_range() function is used which returns std::pair having iterator lower bound (inclusive) and upper bound (exclusive) respectively. If the key does not exist, both iterators would point to end().

```
        auto eqr = mmp.equal_range(6);
        auto st = eqr.first, en = eqr.second;
        for(auto it = st; it != en; ++it){
         std::cout << it->first << ", " << it->second << std::endl;
        }

                // prints:      6, 5

                //              6, 7
```

2.2.2 MultiMaps

Multimaps are part of the C++ STL (Standard Template Library). Multimaps are the associative containers like map that stores sorted key-value pair, but unlike maps which store only unique keys, multimap can have duplicate keys. By default it uses < operator to compare the keys.

For example,

Key	**Value**
1	Vishnu
2	Amal
2	Sudheesh
3	Remya

Syntax,

```
template < class Key,  // multimap::key_type
       // multimap::mapped_type
          class T,

      // multimap::key_compare
          class Compare = less<Key>,
 //multimap::allocator_type
    class Alloc = allocator<pair<const Key,T> >
> class multimap;
```

Functions	Description
begin	Returns an iterator pointing to the first element in the multimap.
cbegin	Returns a const_iterator pointing to the first element in the multimap.
end	Returns an iterator pointing to the past-end.
cend	Returns a constant iterator pointing to the past-end.
rbegin	Returns a reverse iterator pointing to the end.
rend	Returns a reverse iterator pointing to the beginning.
crbegin	Returns a constant reverse iterator pointing to the end.
crend	Returns a constant reverse iterator pointing to the beginning.

begin	Returns an iterator pointing to the first element in the multimap.
cbegin	Returns a const_iterator pointing to the first element in the multimap.
end	Returns an iterator pointing to the past-end.
cend	Returns a constant iterator pointing to the past-end.
rbegin	Returns a reverse iterator pointing to the end.
rend	Returns a reverse iterator pointing to the beginning.

crbegin	Returns a constant reverse iterator pointing to the end.
crend	Returns a constant reverse iterator pointing to the beginning.

insert	Insert element in the multimap.
erase	Erase elements from the multimap.
swap	Exchange the content of the multimap.
clear	Delete all the elements of the multimap.
emplace	Construct and insert the new elements into the multimap.
emplace_hint	Construct and insert new elements into the multimap by hint.

key_comp	Return a copy of key comparison object.
value_comp	Return a copy of value comparison object.

find	Search for an element with given key.
count	Gets the number of elements matching with given key.
lower_bound	Returns an iterator to lower bound.
upper_bound	Returns an iterator to upper bound.
equal_range()	Returns the range of elements matches with given key.

get_allocator	Returns an allocator object that is used to construct the multimap.

2.2.3 Set

An associative container that contains a sorted set of unique objects of type key.

Need to include **<set>** header file

Usually implemented as Red Black Trees

```
template < class T,                          // set::key_type/value_type
           class Compare = less<T>,        // set::key_compare/value_compare
           class Alloc = allocator<T>        // set::allocator_type
           > class set;
```

Functions	Description
Begin	Returns an iterator pointing to the first element in the set.
cbegin	Returns a const iterator pointing to the first element in the set.
End	Returns an iterator pointing to the past-end.
Cend	Returns a constant iterator pointing to the past-end.
rbegin	Returns a reverse iterator pointing to the end.
Rend	Returns a reverse iterator pointing to the beginning.
crbegin	Returns a constant reverse iterator pointing to the end.
Crend	Returns a constant reverse iterator pointing to the beginning.
empty	Returns true if set is empty.
Size	Returns the number of elements in the set.
max_size	Returns the maximum size of the set.
insert	Insert element in the set.
Erase	Erase elements from the set.

Swap	Exchange the content of the set.
Clear	Delete all the elements of the set.
emplace	Construct and insert the new elements into the set.
emplace_hint	Construct and insert new elements into the set by hint.
key_comp	Return a copy of key comparison object.
value_comp	Return a copy of value comparison object.

Find	Search for an element with given key.
count	Gets the number of elements matching with given key.
lower_bound	Returns an iterator to lower bound.
upper_bound	Returns an iterator to upper bound.
equal_range	Returns the range of elements matches with given key.

get_allocator	Returns an allocator object that is used to construct the set.

Set	**Unordered Set**
Set stores elements in a sorted order	Unordered Set stores elements in an unsorted order
Set stores/acquire unique elements only	Unordered Set stores/acquire only unique values
Set uses Binary Search Trees for implementation	Unordered Set uses Hash Tables for implementation
More than one element can be erased by giving the starting and ending iterator	We can erase that element for which the iterator position is given
set<datatype> Set_Name;	unordered_set<datatype> UnorderedSet_Name;

2.2.4 MultiSet

Multisets are a type of associative containers similar to the set, with the exception that multiple elements can have the same values

Function	Definition
begin()	Returns an iterator to the first element in the multiset.
end()	Returns an iterator to the theoretical element that follows the last element in the multiset.
size()	Returns the number of elements in the multiset.
max_size()	Returns the maximum number of elements that the multiset can hold.
empty()	Returns whether the multiset is empty.
pair insert(const g)	Adds a new element ‘g’ to the multiset.
iterator insert (iterator position,const g)	Adds a new element ‘g’ at the position pointed by the iterator.
erase(iterator position)	Removes the element at the position pointed by the iterator.
erase(const g)	Removes the value ‘g’ from the multiset.
clear()	Removes all the elements from the multiset.
key_comp() / value_comp()	Returns the object that determines how the elements in the multiset are ordered (‘<‘ by default).
find(const g)	Returns an iterator to the element ‘g’ in the multiset if found, else returns the iterator to end.
count(const g)	Returns the number of matches to element ‘g’ in the multiset.
lower_bound(const g)	Returns an iterator to the first element that is equivalent to ‘g’ or definitely will not go before the element ‘g’ in the multiset if found, else returns the iterator to end.
upper_bound(const g)	Returns an iterator to the first element that will go after the element ‘g’ in the multiset.
multiset::swap()	This function is used to exchange the contents of two multisets but the sets must be of the same type, although sizes may differ.
multiset::operator=	This operator is used to assign new contents to the container by replacing the existing contents.
multiset::emplace()	This function is used to insert a new element into the multiset container.
multiset equal_range()	Returns an iterator of pairs. The pair refers to the range that includes all the elements in the container which have a key equivalent to k.
multiset::emplace_hint()	Inserts a new element in the multiset.
multiset::rbegin()	Returns a reverse iterator pointing to the last element in the multiset container.
multiset::rend()	Returns a reverse iterator pointing to the theoretical element right

Function	Definition
	before the first element in the multiset container.
multiset::cbegin()	Returns a constant iterator pointing to the first element in the container.
multiset::cend()	Returns a constant iterator pointing to the position past the last element in the container.
multiset::crbegin()	Returns a constant reverse iterator pointing to the last element in the container.
multiset::crend()	Returns a constant reverse iterator pointing to the position just before the first element in the container.
multiset::get_allocator()	Returns a copy of the allocator object associated with the multiset.

To traverse the elements of its phone_book container

```
for (const auto& ele : phone_book)
        cout << ele.first << " can be reached at " << ele.second << endl;
```

Can be written in C++17 as-

```
for (const auto& [name, number] : phone_book)
        cout << name << " can be reached at " << number << endl;
```

2.4 Derived Containers

The STL provides three derived containers namely,

1. Stack
2. Queue
3. Priority_queue.

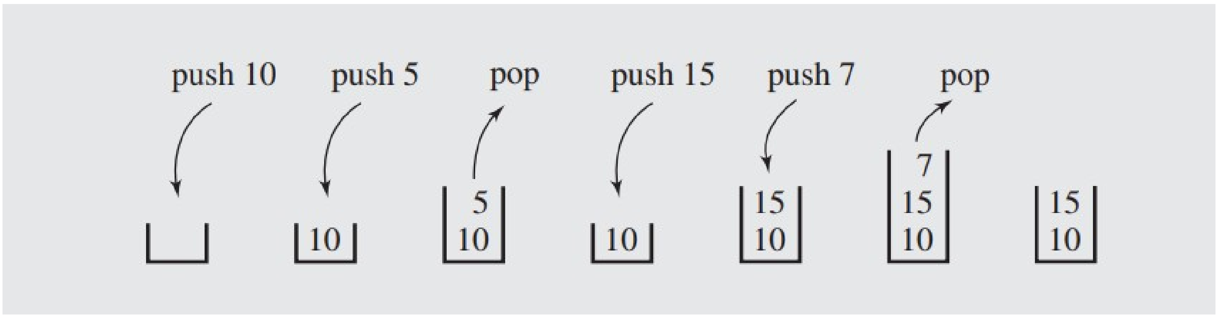
These are also known as **container adaptors**.

Stacks, queues and priority queues can be created from different sequence containers. The derived containers **do not support iterators** and therefore we cannot use them for data manipulation.

However, they support two member functions **pop()** and **push()** for *implementing deleting and inserting operations.*

2.3.1 *Stack*

Stack uses LIFO (Last In First Out) technique. The element which was first inserted will be extracted at the end and so on. There is an element called as 'top' which is the element at the upper most position. All the insertion and deletion operations are made at the **top element** itself in the stack. Stacks in the application areas are implied as the container adaptors.



Different operations

empty
size
back
push_back
pop_back

Syntax:

```
template <class T,
        class Container = deque <T> >
        class stack;
```

2.3.1.1 Stack Functions

Function	Description
(constructor)	The function is used for the construction of a stack container.
empty	The function is used to test for the emptiness of a stack. If the stack is empty the function returns true else false.
size	The function returns the size of the stack container, which is a measure of the number of elements stored in the stack.

top	The function is used to access the top element of the stack. The element plays a very important role as all the insertion and deletion operations are performed at the top element.
push	The function is used for the insertion of a new element at the top of the stack.
pop	The function is used for the deletion of element, the element in the stack is deleted from the top.
emplace	The function is used for insertion of new elements in the stack above the current top element.
swap	The function is used for interchanging the contents of two containers in reference.
relational operators	The non member function specifies the relational operators that are needed for the stacks.
uses allocator<stack>	As the name suggests the non member function uses the allocator for the stacks.

*2.3.1 **Queue***

Queue is an First In First Out (FIFO) structure. Unlike a stack, a queue is a structure in which both ends are used: one for adding new elements and one for removing them. Therefore, the last element has to wait until all elements preceding it on the queue are removed. Elements are inserted at the back (end) and are deleted from the front.

Queues use an encapsulated object of deque or list (sequential container class) as its underlying container, providing a specific set of member functions to access its elements.

Syntax:

```
template<class T,
class Container = deque<T>
> class queue;
```

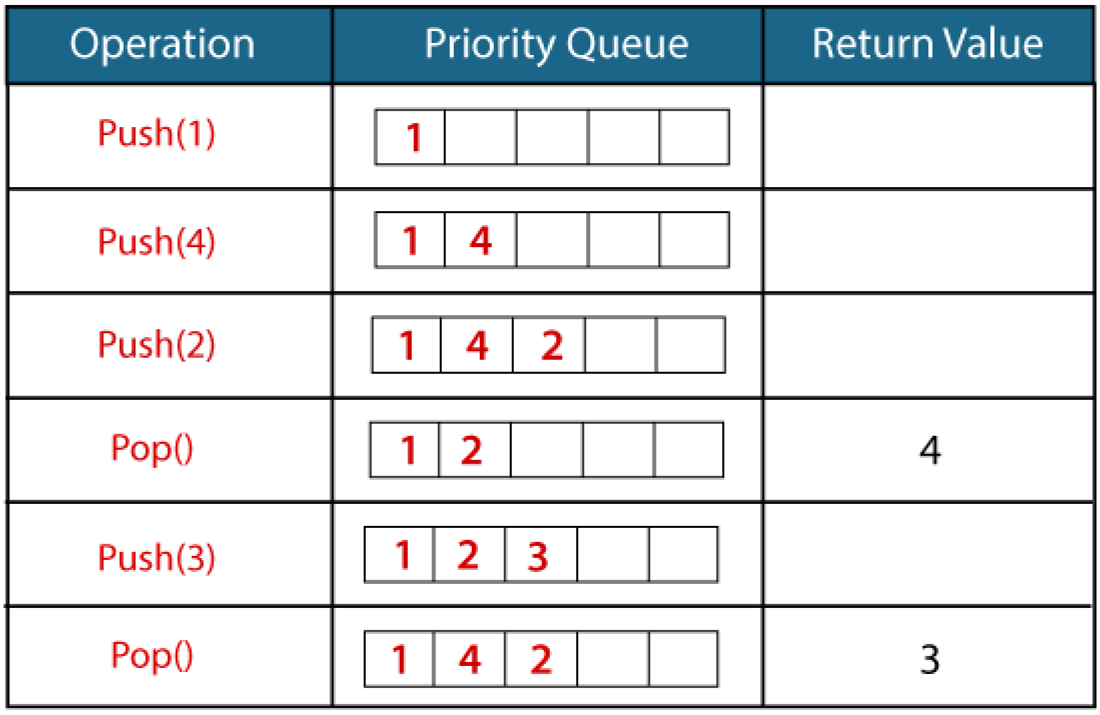
Queue Functions

Function	Description
(constructor)	The function is used for the construction of a queue container.
empty	The function is used to test for the emptiness of a queue. If the

	queue is empty the function returns true else false.
size	The function returns the size of the queue container, which is a measure of the number of elements stored in the queue.
front	The function is used to access the front element of the queue. The element plays a very important role as all the deletion operations are performed at the front element.
back	The function is used to access the rear element of the queue. The element plays a very important role as all the insertion operations are performed at the rear element.
push	The function is used for the insertion of a new element at the rear end of the queue.
pop	The function is used for the deletion of element; the element in the queue is deleted from the front end.
emplace	The function is used for insertion of new elements in the queue above the current rear element.
swap	The function is used for interchanging the contents of two containers in reference.
relational operators	The non member function specifies the relational operators that are needed for the queues.
uses allocator<queue>	As the name suggests the non member function uses the allocator for the queues.

2.3.1 *Priority_Queue*

Priority queue only considers the highest priority element. It pops the elements based on the priority, i.e., the highest priority element is popped first. **By default, the top element is always the greatest element.** We can also change it to the smallest element at the top. Priority queues are built on the top to the max heap and uses array or vector as an internal structure.



Operation	Priority Queue	Return Value
Push(1)	1	
Push(4)	1 4	
Push(2)	1 4 2	
Pop()	1 2	4
Push(3)	1 2 3	
Pop()	1 4 2	3

Function	Description
push()	It inserts a new element in a priority queue.
pop()	It removes the top element from the queue, which has the highest priority.
top()	This function is used to address the topmost element of a priority queue.
size()	It determines the size of a priority queue.
empty()	It verifies whether the queue is empty or not. Based on the verification, it returns the status.
swap()	It swaps the elements of a priority queue with another queue having the same type and size.
emplace()	It inserts a new element at the top of the priority queue.

Algorithms

- Algorithms are functions that can be used generally across a variety of containers for processing their contents.
- STL provides more than sixty standard algorithms to support more extended or complex operations. Standard algorithms also permit us to work with two different types of containers at the same time.
- STL algorithms are not member functions or friends of containers. They are standalone template functions
- STL algorithms reinforce the philosophy of reusability.
- To have access to the STL algorithms, we must include <algorithm> in our program.
- STL algorithms, based on the nature of operations they perform, may be categorized as under:

1. **Retrieve or non-mutating algorithms:** Algorithms in this category are used to process and/or search a container, but never modify the container's elements.
2. **Mutating algorithms:** Algorithms in this category are used to modify containers in some way.
3. **Sorting algorithms:** There are a number of algorithms available for sorting, searching and merging containers and their elements
4. **Set algorithms :** Produce set elements
5. **Relational algorithms:** Used for relational operations
6. **Numeric algorithms:** These types of algorithms are used to perform some kind of mathematical operation against elements in a container.

Non-mutating algorithms

The non-mutating algorithms never modify the containers they are working on.

Sl No.	Operations	Description
1	adjacent_find( )	Finds adjacent pair of objects that are equal
2	count( )	Counts occurrence of a value in a sequence
3	count_if( )	Counts number of elements that matches a predicate
4	equal( )	True if two ranges are the same
5	find( )	Finds first occurrence of a value in a sequence
6	find_end( )	Finds last occurrence of a value in a sequence
7	find_first_of( )	Finds a value from one sequence in another
8	find_if( )	Finds first match of a predicate in a sequence

9	for_each( )	Apply an operation to each element
10	mismatch( )	Finds first elements for which two sequences differ
11	search()	Finds a subsequence within a sequence
12	search_n( )	Finds a sequence of a specified number of similar elements

1. adjacent_find()

It finds the first occurrence of two consecutive elements that are identical and returns an iterator pointing to the first element if identical element exists consecutively otherwise returns an iterator pointing to the last element

Syntax:

```
    adjacent_find(
        ForwardIt first,
        ForwardIt last );
```

Program:

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

using namespace std;

int main(void) {
  vector<int> v = {1, 2, 3, 3, 5};
  auto it = adjacent_find(v.begin(), v.end());
  if (it != v.end())
    cout << "First occurrence of consecutive identical element = " << *it << endl;

  v[3] = 4;

  it = adjacent_find(v.begin(), v.end());
  if (it == v.end())
    cout << "There are no consecutive identical elements.." << endl;
  return 0;
}
```

2. count() - The function returns the count of a value in the range.

Syntax:

```
count(start_point, end_point, value);
```

Program:

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

int main()
{
        int arr[]={2,3,5,3,5,6,3,5,5,5,5,4,5,3,6,7};
        int s =sizeof(arr)/sizeof(arr[0]);
        cout << "Number of times 5 appears :";
        cout << count(arr, arr + s, 5);

        vector<int> n={1,2,3,4,2,2,2,5,5,3,2,2,2,7,2};
        cout << "\nNumber of times 2 appears : ";
        cout<<count(n.begin(), n.end(), 2);

        string str ="thinkpalmtechnologies";
        cout << "\nNumber of times 'n' appears : ";
        cout << count(str.begin(), str.end(), 'n');

        return 0;
}
```

3. count_if() - is used to get the number of elements in a specified range which satisfy a condition. This function returns an integer value which is the number of elements which satisfies the condition.

Syntax:

```
count_if(start, end, condition);
```

Program:

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

```
using namespace std;
bool check_odd(int i){
   if (i % 2!= 0)
      return true;
   else
      return false;
}
 int main()
{
         vector<int> vec;
         for (int i = 0; i < 10; i++){
         vec.push_back(i);
   }

   int total_odd = count_if(vec.begin(), vec.end(), check_odd);
   cout<<"Number of odd is: "<<total_odd;
   return 0;
}
```

4. equal() - which is used to check whether two sequence ranges have the same contents or not.

Syntax:

```
equal (
     InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2)
```

Program:

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

int main()
{
    vector<int> arr1{ 3, 2, 1, 4, 5, 6, 7 };

    vector<int> arr2{ 3, 2, 1, 4, 5 };
      if (equal(arr1.begin(), arr1.begin() + 5, arr2.begin())) {
        cout << "both ranges are exactly equal \n";
    }
    else
        cout << "both ranges are not exactly equal \n";
        vector<int> arr3{ 1, 2, 3, 4, 5, 6, 7 };
        vector<int> arr4{ 1, 2, 3, 4, 5 };
    if (equal(arr3.begin(), arr3.end(), arr4.begin())) {
        cout << "both ranges are exactly equal \n";
    }
    else
        cout << "both ranges are not exactly equal \n";

    return 0;
}
```

5. find() - the iterator starts the search from the first element and goes on to the last element, if the element is found in the range then it is returned otherwise the last element of the range is given.

Syntax:

```
find (
InputIterator first,
InputIterator last,
const T& val)
```

Program:

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

using namespace std;

int main(void)
{

   int val = 5;
   vector<int> v = {1, 2, 3, 4, 5};
   auto result = find(v.begin(), v.end(), val);

   if (result != end(v))
      cout << "Vector contains element " << val << endl;
   val = 15;

   result = find(v.begin(), v.end(), val);
```

```
  if (result == end(v))
    cout << "Vector doesn't contain element " << val << endl;
  return 0;
}
```

6. find_end( ): Finds last occurrence of a value in a sequence

Syntax:

```
    find_end (
        ForwardIterator1 first1,
        ForwardIterator1 last1,
        ForwardIterator2 first2,
        ForwardIterator2 last2);
```

Program:

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

using namespace std;

int main(void) {
  vector<int> v1 = {1, 2, 1, 2, 1, 2};
  vector<int> v2 = {1, 2};

  auto result = find_end(v1.begin(), v1.end(), v2.begin(), v2.end());

  if (result != v1.end())
    cout << "Last sequence found at location "
      << distance(v1.begin(), result) << endl;

  v2 = {1, 3};

  result = find_end(v1.begin(), v1.end(), v2.begin(), v2.end());

  if (result == v1.end())
    cout << "Sequence doesn't present in vector." << endl;

  return 0;
}
```

7. find_first_of( ):Finds a value from one sequence in another

Syntax:

```
find_first_of(
     ForwardIterator1 first1,
     ForwardIterator1 last1,
     ForwardIterator2 first2,
     ForwardIterator2 last2);
```

Program:

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

int main(void) {
  vector<int> v1 = {5, 2, 6, 1, 3, 4, 7};
  vector<int> v2 = {10, 1};

  auto result = find_first_of(v1.begin(), v1.end(), v2.begin(), v2.end());

  if (result != v1.end())
    cout << "Found first match at location "
      << distance(v1.begin(), result) << endl;

  v2 = {11, 13};
  result = find_end(v1.begin(), v1.end(), v2.begin(), v2.end());

  if (result == v1.end())
    cout << "Sequence doesn't found." << endl;

  return 0;
}
```

8. find_if( ):Finds first match of a predicate in a sequence

Syntax:

```
find_if (
     InputIterator first,
     InputIterator last,
          UnaryPredicate pred);
```

Program:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
bool unary_pre(int n) {
  return ((n % 2) == 0);
}
int main() {
  vector<int> v = {10, 2, 3, 4, 5};
  auto it = find_if(v.begin(), v.end(), unary_pre);
  if (it != end(v))
    cout << "First even number is " << *it << endl;
  v = {1};
  it = find_if(v.begin(), v.end(), unary_pre);
  if (it == end(v))
    cout << "Only odd elements present in the sequence." << endl;
  return 0;
}
```

9. for_each( ): This loop accepts a function which executes over each of the container elements. This loop is defined in the header file “algorithm”:

Syntax:

```
     for_each (
          InputIterator start_iter,
          InputIterator last_iter,
          Function fnc)
```

Program:

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

using namespace std;

int print_even(int n) {
  if (n % 2 == 0)
    cout << n << ' ';
}

int main() {
  vector<int> v = {1, 2, 3, 4, 5};

  cout << "Vector contains following even number" << endl;

  for_each(v.begin(), v.end(), print_even);

  cout << endl;

  return 0;
}
```

10. mismatch( ): The function returns a pair pointer where the first element of the pair points to the position of the first mismatched element of the first container and the second element of the pair points to the position of the first mismatched element of the second container. If no mismatch is found, then the resulting pair points to the position after the last element of the first container and the position of the corresponding element in the second container, respectively.

Syntax:

```
    mismatch(
        start_iter1,
        end_iter1,
        start_iter2 )
```

Program:

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

int main()
{

        vector<int> v1 = { 1, 10, 15, 20 };
        vector<int> v2 = { 1, 10, 25, 30, 45 };
        vector<int> v3 = { 1, 10, 15, 20 };
        vector<int> v4 = { 1, 10, 15, 20, 24 };

        pair< vector<int>::iterator,
        vector<int>::iterator > mispair;

        mispair = mismatch(v1.begin(), v1.end(), v2.begin());

        cout << "The 1st mismatch element of 1st container : ";
        cout << *mispair.first << endl;

        cout << "The 1st mismatch element of 2nd container : ";
        cout << *mispair.second << endl;

        mispair = mismatch(v3.begin(), v3.end(), v4.begin());

        cout << "The returned value from 1st container is : ";
        cout << *mispair.first << endl;

        cout << "The returned value from 2nd container is : ";
        cout << *mispair.second << endl;

}
```

11. search(): Finds a subsequence within a sequence

Syntax:

```
        search (
            ForwardIterator1 first1,
            ForwardIterator1 last1,
            ForwardIterator2 first2,
            ForwardIterator2 last2);
```

Program:

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

using namespace std;

int main()
{
        int i, j;

        vector<int> v1 = { 1, 2, 3, 4, 5, 6, 7 };
        vector<int> v2 = { 3, 4, 5 };
        vector<int>::iterator i1;

        i1 = search(v1.begin(), v1.end(), v2.begin(), v2.end());

        if (i1 != v1.end()) {
                cout << "vector2 is present at index " << (i1 - v1.begin());
        } else {
                cout << "vector2 is not present in vector1";
        }

        return 0;
}
```

12. search_n( ):Finds a sequence of a specified number of similar elements

Syntax:

```
        search_n (
                ForwardIterator first,
                ForwardIterator last,
                Size count,
                const T& val);
```

Program:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
int i, j;
vector<int> v1 = { 1, 2, 3, 4, 5, 3, 3, 6, 7 };
int v2 = 3;

vector<int>::iterator i1;
i1 = search_n(v1.begin(), v1.end(), 2, v2);

if (i1 != v1.end())
        cout << "v2 is present consecutively 2 times at index "<< (i1 - v1.begin());
else
        cout << "v2 is not present consecutively 2 times in "<< "vector v1";
        return 0;
}
```

Mutating algorithms

These algorithms are modifying algorithms that are designed to work on the container elements and perform operations like shuffle, rotation, changing, etc.

Sl No	Operations	Description
1	Copy( )	Copies a sequence
2	copy_backward( )	Copies a sequence from the end
3	fill( )	Fills a sequence with a specified value
4	fill_n( )	Fills first n elements with a specified value
5	generate( )	Replaces all elements with the result of an operation
6	generate_n( )	Replaces first n elements with the result of an operation
7	iter_swap( )	Swaps elements pointed to by iterators
8	random_shuffle( )	Places elements in random order
9	remove( )	Deletes elements of a specified value
10	remove_copy( )	Copies a sequence after removing a specified value
11	remove_copy_if( )	Copies a sequence after removing elements matching a predicate
12	remove_if( )	Deletes elements matching a predicate
13	replace( )	Replaces elements with a specified value
14	replace_copy( )	Copies a sequence replacing elements with a given value
15	replace_copy_if( )	Copies a sequence replacing elements matching a predicate
16	replace_if( )	Replaces elements matching a predicate
17	reverse( )	Reverses the order of elements
18	reverse_copy( )	Copies a sequence into reverse order
19	rotate( )	Rotates elements
20	rotate_copy( )	Copies a sequence into a rotated
21	swap( )	Swaps two elements
22	swap_ranges( )	Swaps two sequences
23	transform( )	Applies an operation to all elements
24	unique( )	Deletes equal adjacent elements
25	unique_copy( )	Copies after removing equal adjacent elements

1. Copy( ): it is used to copy the elements of a container, it copies the elements of a container from given range to another container from a given beginning position.

Syntax:

```
copy(
     strt_iter1,
     end_iter1,
     strt_iter2);
```

Program:

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

int main()
{
    int arr[] = { 10, 20, 30, 40, 50 };
    vector<int> v1(5);
    copy(arr, arr + 5, v1.begin());
    cout << "arr: ";
    for (int x : arr)
        cout << x << " ";
    cout << endl;

    cout << "v1: ";
    for (int x : v1)
        cout << x << " ";
    cout << endl;

    return 0;
}
```

2. copy_backward( ): This function starts copying elements into the destination container from backward and keeps on copying till all numbers are not copied. The copying starts from the “strt_iter2” but in the backward direction. It also takes similar arguments as copy().

Syntax:

```
copy_backward(
     BidirectionalIterator1 first,
     BidirectionalIterator1 last,
     BidirectionalIterator2 result);
```

Program:

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

int main()
{

vector<int> v1 = { 1, 5, 7, 3, 8, 3 };
vector<int> v2(6);
vector<int> v3(6);

copy(v1.begin(), v1.begin()+3, v2.begin());

cout << "The new vector elements entered using copy() : ";
for(int i=0; i<v2.size(); i++)
cout << v2[i] << " ";
cout << endl;

copy_n(v1.begin(), 4, v3.begin());

cout << "The new vector elements entered using copy_n() : ";
for(int i=0; i<v3.size(); i++)
cout << v3[i] << " ";

}
```

3. fill(): it is used to assign a value to the all elements within a given range of a container, it accepts iterators pointing to the starting and ending position in the container and a value to be assigned to the elements within the given range, and assigns the value.

Syntax:
```
      fill(iterator start, iterator end, value);
```

Program:
```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
  vector<int> v(10);

  fill(v.begin(), v.end(), -1);
  cout << "v: ";
  for (int x : v)
    cout << x << " ";
  cout << endl;

  fill(v.begin(), v.begin() + 2, 50);

  cout << "v: ";
  for (int x : v)
    cout << x << " ";
  cout << endl;

  fill(v.begin() + 2, v.end(), 100);
  cout << "v: ";
  for (int x : v)
    cout << x << " ";
  cout << endl;

  return 0;
}
```

4. fill_n( ): it is used to assign a value to the n elements of a container, it accepts an iterator pointing to the starting position in the container, n (number of elements) and a value to be assigned to the n elements, and assigns the value.

Syntax:

```
fill_n(iterator start, n, value);
```

Program:

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

int main()
{
    vector<int> v(10);

    fill_n(v.begin(), 10, -1);

    cout << "v: ";
    for (int x : v)
        cout << x << " ";
    cout << endl;

    fill_n(v.begin(), 3, 100);

    cout << "v: ";
    for (int x : v)
        cout << x << " ";
    cout << endl;

    fill_n(v.begin() + 3, 7, 200);

    cout << "v: ";
    for (int x : v)
        cout << x << " ";
    cout << endl;

    return 0;
}
```

5. generate( ): It is used to generate numbers based upon a generator function, and then, it assigns those values to the elements in the container in the range [first, last)

Syntax:

```
    generate (
        ForwardIterator first,
        ForwardIterator last,
        Generator gen);
```

Program:

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

int gen()
{
    static int i = 0;
    return ++i;
}

using namespace std;
int main()
{
    int i;
    vector<int> v1(10);
    generate(v1.begin(), v1.end(), gen);

    vector<int>::iterator i1;
    for (i1 = v1.begin(); i1 != v1.end(); ++i1) {
        cout << *i1 << " ";
    }
    return 0;
}
```

6. generate_n( ): This function does the same task as the generate function. The only difference is that in generate() function the second argument is the Forward Iterator to the last position whereas in generate_n() function it is the number of values that are to be generated.

Syntax:

```
    generate_n (
        OutputIterator first,
        Size n,
        Generator gen);
```

Program:

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

using namespace std;

int main()
{
  srand(time(0));
  vector<int> vec(10);
  generate_n(vec.begin(), 10, rand);
  cout << "Generated elements are:\n";
  for(int i=0; i<10; i++)
  cout << vec[i] << endl;
  return 0;
}
```

7. iter_swap( ): It exchange values of objects pointed by two iterators. It uses function swap (unqualified) to exchange the elements.

Syntax:

```
iter_swap (
     ForwardIterator1 a,
     ForwardIterator2 b);
```

Program:

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

using namespace std;

int main(void) {
  vector<int> v1 = {1, 2, 3, 4, 5};
  vector<int> v2 = {10, 20, 30, 40, 50};

  iter_swap(v1.begin(), v2.begin());
  iter_swap(v1.begin() + 1, v2.begin() + 2);

  cout << "Vector v2 contains following elements" << endl;

  for (auto it = v2.begin(); it != v2.end(); ++it)
    cout << *it << endl;

  return 0;
}
```

8. random_shuffle( ): It is used to randomly rearrange the elements in range [left, right). This function randomly swaps the positions of each element with the position of some randomly chosen positions.

Syntax:

```
random_shuffle (
     RandomAccessIterator first,
     RandomAccessIterator last);
```

Program:

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

int main() {
  vector<int> vec = {1,2,3,4,5,6,7,8,9};

  random_shuffle(vec.begin(), vec.end());

  cout << "Updated vector: ";
  for (int x: vec)
    cout << x << " ";
  return 0;
}
```

9. remove( ):Deletes elements of a specified value

Syntax:

```
          remove  (
               ForwardIterator first,
               ForwardIterator last,
               const T& val)
```

Program:

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

using namespace std;

int main () {
 int myints[] = {10,20,30,50,20,40,100,20};

 int* pbegin = myints;
 int* pend = myints+sizeof(myints)/sizeof(int);

 pend = remove (pbegin, pend, 20);
 cout << "range contains:";
 for (int* p=pbegin; p!=pend; ++p)
 cout << ' ' << *p;
 cout << '\n';
 return 0;
}
```

10. remove_copy( ):Copies a sequence after removing a specified value

Syntax:

```
          remove_copy (
               InputIterator first,
               InputIterator last,
               OutputIterator result,
               const T& val)
```

Program:

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

void print(vector<int>&v)
{
int len = v.size();
for (int i = 0; i < len; i++)
cout << v[i] << " ";
cout << endl;
}

int main()
{

vector <int> v1, v2(10);

for(int i = 10; i <= 25; i++)
v1.push_back(i % 6);

cout << "elements of v1 before remove_copy: "<<endl;
print(v1);
remove_copy(v1.begin(), v1.end(), v1.begin(), 3);
cout << "After removing element 3" <<endl;
print(v1);
return 0;
}
```

11. remove_copy_if( ):Copies a sequence after removing elements matching a predicate

Syntax:

```
remove_copy_if (
          InputIterator first,
          InputIterator last,
          OutputIterator result,
          UnaryPredicate pred);
```

Program:

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
bool IsOdd(int i)
{
     return ((i % 2) != 0);
 }

void print(vector<int>&v)
{
int len = v.size();
for (int i = 0; i < len; i++)
cout << v[i] << " ";
cout << endl;
}

int main()
{

vector <int> v1, v2(10);
for(int i = 10; i <= 20; i++)
v1.push_back(i);
cout << "elements of v1 before remove_copy: ";
print(v1);
remove_copy_if(v1.begin(), v1.end(), v2.begin(), IsOdd);
cout << "elements of v1 after remove_copy: ";
print(v1);
cout << "After removing Odd Numbers from v1"
" copy result in vector v2" <<endl;
print(v2);
return 0;
}
```

12. remove_if( ):Deletes elements matching a predicate

Syntax:

```
          remove_if (
                    ForwardIterator first,
                    ForwardIterator last,
                    UnaryPredicate pred);
```

Program:

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
int main()
{
vector<int> vec1{10, 20, 30, 30, 20, 10, 10, 20};
vector<int> vec2{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
cout << "Original vector : ";
for (int i = 0; i < vec1.size(); i++)
cout << " " << vec1[i];
cout << "\n";
vector<int>::iterator pend;
pend = remove(vec1.begin(), vec1.end(), 20);
cout << "After remove : ";
for (vector<int>::iterator p = vec1.begin();
p != pend; ++p)
cout << ' ' << *p;
cout << '\n';
cout << "\nOriginal vector : ";
for (int i = 0; i < vec2.size(); i++)
cout << " " << vec2[i];
cout << "\n";
pend = remove_if(vec2.begin(), vec2.end(), IsOdd);
pend = remove_if(
vec2.begin(), vec2.end(),
[](int i) { return ((i % 2) == 1); });
cout << "After remove_if : ";
for (vector<int>::iterator q = vec2.begin();
q != pend; ++q)
cout << ' ' << *q;
cout << '\n';
return 0;
}
```

13. replace( ): it is used to replace an old value with a new value in the given range of a container, it accepts iterators pointing to the starting and ending positions, an old value to be replaced and a new value to be assigned.

Syntax:

```
 replace(
     iterator start,
     iterator end,
     const T& old_value,
     const T& new_value);
```

Parameters:

`iterator start, iterator end` – these are the iterators pointing to the starting and ending positions in the container, where we have to run the replace operation.
`old_value` – is the value to be `searched` and replaced with the new value.
`new_value` – a value to be assigned instead of an old_value.

Program:

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

int main()
{
        string s1 = "HeyWorld !";
        s1.replace(0, 3, "Hello ");
        cout << s1 << endl;
        return 0;
}
```

14. replace_copy( ): it is used to replace value in the range and copy the elements in the result with replaced values, it accepts a range [start, end] of the sequence, beginning position of the result sequence, old and new value and it replaces the all old_value with the new_value in the range and copies the range to the range beginning at result.

Syntax:

```
 replace_copy(
        iterator start,
        iterator end,
        iterator start_result,
        const T& old_value,
        const T& new_value);
```

Parameters:

`iterator start, iterator end` – iterators pointing to the starting and ending positions in the container, where we have to run the replace operation.
`iterator start_result` – It is the beginning iterator of the result sequence.
`old_value` – a value to be replaced.
`new_value` – a value to be assigned instead of an old_valu

Program:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void print(vector<char>& v)
{
int len = v.size();
for (int i = 0; i < len; i++)
cout << v[i] << " ";
cout << endl;
}

int main()
{
	vector<char> v;


  for (int i = 0; i <= 6; i++)
		v.push_back('A' + i);

  cout << "Before replace_copy " <<": ";
	print(v);

	replace_copy(v.begin(), v.begin()+1,  v.begin(), 'A', 'Z' );
```

```
    cout << "After replace_copy " << ": ";
        print(v);

        return 0;
}
```

15. replace_copy_if( ): it is used to replace value in the given range and copy the elements in the result sequence with replaced values, it accepts a range [start, end] of the sequence, beginning position of the result sequence, a unary function (that will be used to validate the elements) and new value, it replaces the all elements which pass the test case applied in unary function (i.e. if function returns true) with the new_value in the range and copies the range to the beginning at result sequence.

Syntax:

```
 replace_copy_if(
        iterator start,
        iterator end,
        iterator start_result,
        unary_function,
        const T& new_value);
```

Parameters:

`iterator start, iterator end` – these are the iterators pointing to the starting and ending positions in the container, where we have to run the replace operation.
`iterator start_result` – is the beginning iterator of the result sequence.
`unary_function` – is a unary function that will perform the condition check on all elements in the given range and the elements which pass the condition will be replaced.
`new_value` – a value to be assigned instead of an old_value.

Program:

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

int IsOdd(int i)
{
```

```
return ((i % 2) != 0);

}

void print(vector<int>& v)
{
int len = v.size();
for (int i = 0; i < len; i++)
cout << v[i] << " ";
cout << endl;
}
int main()
{


    vector<int> v1, v2;
          for (int i = 1; i <= 10; i++)
        v1.push_back(i);

          cout << "Before replace_copy_if : ";
          print(v1);

          v2.resize(v1.size());
    replace_copy_if(v1.begin(), v1.end(),v2.begin(), IsOdd, 0);
    cout << "After replace_copy_if : ";
          print(v2);
          return 0;
}
```

16. replace_if( ): it is used to replace the value in a given range based on the given unary function that should accept an element in the range as an argument and returns the value which should be convertible to bool (like 0 or 1), it returns value indicates whether the given elements can be replaced or not?

Syntax:

```
replace_if(
        iterator start,
        iterator end,
        unary_function,
        const T& new_value);
```

Parameters:

`iterator start, iterator end` – these are the iterators pointing to the starting and ending positions in the container, where we have to run the replace operation.

`unary_function` – is a unary function that will perform the condition check on all elements in the given range and the elements which pass the condition will be replaced.

`new_value` – a value to be assigned instead of an old_value.

Program:

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

bool IsVowel(char ch)
{
return (ch == 'A' || ch == 'E' || ch == 'I' || ch == 'O' || ch == 'U');
}

void print(vector<char>&v)
{
int len = v.size();
for (int i = 0; i < len; i++)
cout << v[i] << " ";
cout << endl;
}

int main()
{
vector<char> v;
for (int i = 0; i <= 6; i++)
v.push_back('A' + i);

cout << "Before replace_if " <<": ";
print(v);

replace_if(v.begin(), v.end(), IsVowel, 'V');

cout << "After replace_if " << ": ";
print(v);
return 0;
}
```

17. reverse( ): It reverses the order of the elements in the range [first, last) of any container.

Syntax:

```
 reverse(
        iterator start,
        iterator end);
```

Parameters:

`iterator start, iterator end` – these are the iterators pointing to the starting and ending positions in the container, where we have to run the reverse operation.

Program:

```
int main()
{

     vector<int> vec1;
     for (int i = 0; i < 8; i++)
        vec1.push_back(i + 10);


   cout<<"Printing elements before reverse\n";
   print(vec1);

   reverse(vec1.begin() + 5, vec1.begin() + 8);

   cout<<"Printing elements after reverse\n";
   print(vec1);
   return 0;
}
```

18. reverse_copy( ): Copies a sequence into reverse order

Syntax:

```
 reverse_copy(
        iterator start,
        iterator end,
        iterator start_result);
```

Parameters:

`iterator start, iterator end` – these are the iterators pointing to the starting and ending positions in the container, where we have to run the replace operation.
`iterator start_result` – is the beginning iterator of the result sequence.

Program:

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

void print(vector<int>&v)
{

    int len = v.size();
    for (int i = 0; i < len; i++)
    cout << v[i] << " ";
    cout << endl;
}

int main()
{

vector<int> vec1;
for (int i = 0; i < 10; i++)
        vec1.push_back(i + 10);

cout<<"Printing vector-1 before reverse\n";
print(vec1);
vector<int> vec2(vec1.size());   //Allocate size

reverse_copy(vec1.begin(), vec1.end(), vec2.begin());

cout<<endl;
cout<<"Printing vector-1 after reverse\n";
print(vec1);
cout<<endl;
cout<<"Printing vector-2 after reverse\n";
print(vec2);

return 0;
}
```

19. rotate( ): It rotates the order of the elements in the range [first, last], in such a way that the element pointed by the middle becomes the new first element.

Syntax:

```
    rotate(
        iterator start,
        iterator middle,
        iterator end);
```

Parameters:

`iterator start` – an iterator pointing to the first element of the sequence.
`iterator middle` – an iterator pointing to the middle or any other elements from where we want to start the rotation.
`iterator end` – an iterator pointing to the last element of the sequence.

Program:

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

void print(vector<int>&v)
{

   int len = v.size();
   for (int i = 0; i < len; i++)
   cout << v[i] << " ";
   cout << endl;
}

int main()
{

vector<int> vec1;
for (int i = 0; i < 10; i++)
        vec1.push_back(i + 10);
cout<<"Printing vector-1 before rotate\n";
print(vec1);
rotate(vec1.begin(), vec1.begin()+3, vec1.end());
cout << "New vector after left rotation :\n";
print(vec1);
vector<int> vec2;
for (int i = 0; i < 10; i++)
        vec2.push_back(i + 10);
cout<<endl;
cout<<"Printing vector-2 before rotate\n";
print(vec2);
rotate(vec2.begin(), vec2.begin()+vec2.size()-4, vec2.end());
print(vec2);
return 0;
}
```

20. rotate_copy( ): it is used to rotate left the elements of a sequence within a given range and copy the rotating elements to another sequence, it accepts the range (start, end) of the input sequence, a middle point, and an iterator pointing to start element of result sequence. It rotates the elements in such a way that the element pointed by the middle iterator becomes the new first element.

Syntax:

```
    rotate_copy(
           iterator start,
            iterator middle,
        iterator end,
        iterator start_result);
```

Parameters:

`iterator start` – an iterator pointing to the first element of the sequence.

`iterator middle` – an iterator pointing to the middle or any other elements from where we want to start the rotation.

`iterator end` – an iterator pointing to the last element of the sequence.

`iterator start_result` – an iterator pointing to the first element in result sequence

Program:

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

void print(vector<int>&v)
{

    int len = v.size();
    for (int i = 0; i < len; i++)
    cout << v[i] << " ";
    cout << endl;
}

int main()
{
vector<int> vec1;
for (int i = 0; i < 10; i++)
vec1.push_back(i + 10);
vector<int> vec2(vec1.size());
cout<<"Printing vector-1 before rotation\n";
print(vec1);
rotate_copy(vec1.begin(), vec1.begin()+2, vec1.end(), vec2.begin());
cout<<"Printing vector-2 after rotation\n";
print(vec2);
return 0;
}
```

21. swap( ): swaps or say interchanges the values of two containers under reference.

Syntax:

```
void swap (T& a, T& b);
```

Program:

```
#include <iostream>
using namespace std;
int main()
```

```
{
    int a = 10, b = 20;
    cout << "Before swapping:\n";
    cout << "a: " << a << ", b: " << b << endl;
    swap(a, b);
    cout << "After swapping:\n";
    cout << "a: " << a << ", b: " << b << endl;
return 0;
}
```

22. swap_ranges( ): swaps elements between two ranges.

Syntax:

```
ForwardIterator swap_ranges(
    ForwardIterator first1,
    ForwardIterator last1,
    ForwardIterator first2
    );
```

Parameters:

first1 : Iterator to beginning of a range.
last1 : Iterator to end of a range.
first2 : Iterator to beginning of another range.

Program:

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
void print(vector<int>&v)
{

    int len = v.size();
    for (int i = 0; i < len; i++)
    cout << v[i] << " ";
    cout << endl;
}
int main()
{
        vector<int> v1;
        for (int i = 0; i < 10; ++i)
        v1.push_back(i);
        cout<<"Vector-1 before swapping\n";
        print(v1);
        vector<int> v2(10, 10);
        cout<<"Vector-2 before swapping\n";
        print(v2);
        cout << "\n";
        swap_ranges(v1.begin() + 3, v1.begin() + 7, v2.begin());
        cout<<"Vector-1 after swapping\n";
        print(v1);
        cout<<"Vector-2 after swapping\n";
        print(v2);
        return 0;
}
```

23. transform( ): Applies an operation to all elements. It has Unary operation mode and Binary operation mode

1. **Unary Operation** : Applies a unary operator on input to convert into output

Syntax:

```
transform(
        Iterator inputBegin,
        Iterator inputEnd,
        Iterator OutputBegin,
        unary_operation)
```

Program:

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

int square(int x)
{
  return x*x;
}
int main()
{
  int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int res[10];
  cout<<"Array before transform operation\n";
    for(int i = 0; i<10; i++)
     cout <<arr[i]<< " ";
   cout<<"\n";
  transform(arr, arr+10, res, square);
   cout<<"Array after transform operation\n";
   for(int i = 0; i<10; i++)
     cout <<res[i]<< " ";

}
```

2. **Binary Operation** : Applies a binary operator on input to convert into output

Syntax:

```
transform(
     Iterator inputBegin1,
     Iterator inputEnd1,
     Iterator inputBegin2,
     Iterator OutputBegin,
     binary_operation)
```

Program:

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

int multiply(int x, int y)
{
   return x*y;
}

int main()
{
    int arr1[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int arr2[10] = {38, 21, 32, 65, 58, 74, 21, 84, 20, 35};
    int res[10];

    cout<<"Array-1 before transform operation\n";
    for(int i = 0; i<10; i++)
      cout<<arr1[i]<<" ";
    cout<<"\n";
    cout<<"Array-2 before transform operation\n";
    for(int i = 0; i<10; i++)
      cout<<arr2[i]<<" ";
    cout<<"\n";
    cout<<"Result after transform operation\n";
    transform(arr1, arr1+10, arr2, res, multiply);
    for(int i = 0; i<10; i++)
        cout<<res[i]<<" ";
    return 0;
}
```

24. unique( ): It does not delete all the duplicate elements, but it removes duplicacy by just replacing those elements by the next element present in the sequence which is not duplicate to the current element being replaced. All the elements which are replaced are left in an unspecified state.

Syntax:

```
unique(
     Iterator inputBegin1,
     Iterator inputEnd1);
```

Program:

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
void print(vector<int>&v)
{

    int len = v.size();
    for (int i = 0; i < len; i++)
    cout << v[i] << " ";
    cout << endl;
}
int main()
{

        vector<int> v1={1,1,2,2,3,2};
    cout<<"Vector-1 before swapping\n";
    print(v1);
    vector<int>::iterator ip;
    ip = unique(v1.begin(),v1.end());
    v1.resize(distance(v1.begin(), ip));
    print(v1);
```

```
        return 0;
}
```

25. unique_copy( ): only the first element from every consecutive group of equivalent elements in the range [first, last) is copied.

Syntax:

```
unique_copy(
          iterator start,
          iterator middle,
          iterator end,
          iterator start_result);
```

Parameters:

`iterator start` – an iterator pointing to the first element of the sequence.
`iterator middle` – an iterator pointing to the middle or any other elements from where we want to start the rotation.
`iterator end` – an iterator pointing to the last element of the sequence.
`iterator start_result` – an iterator pointing to the first element in result sequence

Program:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
        vector<int> v1 = { 10, 10, 30, 30, 30, 100, 10,
                                        300, 300, 70, 70, 80 };
        vector<int> v2(10);
        vector<int>::iterator ip;

        ip = unique_copy(v1.begin(), v1.begin() + 12, v2.begin());
        v2.resize(distance(v2.begin(), ip));

        cout << "Before: ";
        for (ip = v1.begin(); ip != v1.end(); ++ip)
                cout << *ip << " ";

        cout << "\nAfter: ";
        for (ip = v2.begin(); ip != v2.end(); ++ip)
                cout << *ip << " ";

        return 0;
}
```

Sorting algorithms

Sl No	Operations	Description
1	sort( )	Sorts a sequence
2	stable_sort( )	Sorts maintaining order of equal elements
3	partial_sort( )	Sorts a part of a sequence
4	partial_sort_copy()	Sorts a part of a sequence and then copies
5	is_sorted	Checks whether the range is sorted or not.
6	is_sorted_until	Checks till which element a range is sorted.
7	nth_element( )	The functions sorts the elements in the range.
8	merge( )	Merges two sorted sequences
9	inplace_merge( )	Merges two consecutive sorted sequences

1. sort(): it is used to sort the elements in the range [first, last) into ascending order.

Syntax:

```
sort (
         ForwardIterator first,
         ForwardIterator last,
         Compare comp);
```

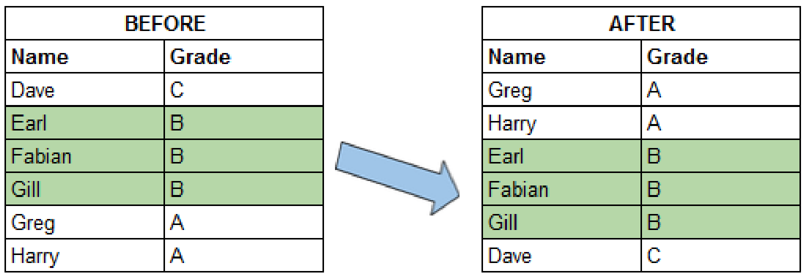
Parameters:

`iterator start` – an iterator pointing to the first element of the sequence.
`iterator end` – an iterator pointing to the last element of the sequence.
`comp` – A user-defined binary predicate function that accepts two arguments and returns true if the two arguments are in order and false otherwise

Program:

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

using namespace std;

int main()
{
  vector<int> v = {3, 1, 4, 2, 5};

    cout<<"Before sorting: ";
    for_each(v.begin(), v.end(), [](int x) {
    cout << x << " ";
  });

  sort(v.begin(), v.end());

  cout<<"\nAfter sorting:  ";
  for_each(v.begin(), v.end(), [](int x) {
    cout << x << " ";
  });

  return 0;
}
```

2. stable_sort() function is used to sort the elements in the range [first, last) into ascending order like sort but keeps the order of equivalent elements.



BEFORE	
Name	**Grade**
Dave	C
Earl	B
Fabian	B
Gill	B
Greg	A
Harry	A

AFTER	
Name	**Grade**
Greg	A
Harry	A
Earl	B
Fabian	B
Gill	B
Dave	C

Syntax:

```
stable_sort (
          ForwardIterator first,
          ForwardIterator last,
          Compare comp);
```

Parameters:

`iterator start` – an iterator pointing to the first element of the sequence.
`iterator end` – an iterator pointing to the last element of the sequence.
`comp` – A user-defined binary predicate function that accepts two arguments and returns true if the two arguments are in order and false otherwise

Program:

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

using namespace std;

int main()
{
  vector<int> v = {3, 1, 4, 2, 5};

    cout<<"Before sorting: ";
    for_each(v.begin(), v.end(), [](int x) {
    cout << x << " ";
  });

  stable_sort(v.begin(), v.end());

  cout<<"\nAfter sorting:  ";
  for_each(v.begin(), v.end(), [](int x) {
    cout << x << " ";
  });

  return 0;
}
```

3. partial_sort() function is used to rearrange the elements in the range[first, last), in such a way that the elements between the first and middle will be sorted and the elements between the middle and last will be in an unspecified order.

Syntax:

```
partial_sort (
          ForwardIterator first,
          ForwardIterator last,
          Compare comp);
```

Parameters:

iterator start – an iterator pointing to the first element of the sequence.
iterator end – an iterator pointing to the last element of the sequence.

Middle – A random access iterator pointing to the one past the final element in the sub-range to be sorted.

Program:

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

bool myfunction (int i,int j)
{
        return (i<j);
}

int main () {
        int myints[] = {9,8,7,6,5,4,3,2,1};
        vector<int> myvector (myints, myints+9);
        partial_sort (myvector.begin(), myvector.begin()+5, myvector.end());
        cout << "myvector contains:";
        for (vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
                cout << ' ' << *it;
        cout << '\n';
        return 0;
}
```

4. partial_sort_copy( ): It is similar to partial_sort() function which is used to rearrange the elements in the range[first, last), in such a way that the elements between the first and middle will be sorted and the elements between middle and last will be in an unspecified order. But partial_sort_copy() function puts the result in a new range[result_first, result_last).

Syntax:

```
partial_sort_copy (
          ForwardIterator first,
          ForwardIterator last,
          result_first,
          result_last);
```

Parameters:

`first`: Input iterator to the first element in the container.
`last`: Input iterator to the last element in the container.

`result_first`: Random-Access iterator pointing to the initial position in the destination container.
`result_last`: Random-Access iterator pointing to the final `position` in the destination container.

Program:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
	vector<int> v = { 1, 1, 3, 10, 3, 3, 7, 7, 8 }, v1(3);
	vector<int>::iterator ip;
	partial_sort_copy(v.begin(), v.end(), v1.begin(), v1.end());

	for (ip = v1.begin(); ip != v1.end(); ++ip) {
		cout << *ip << " ";
	}

	return 0;
}
```

5. merge() - This function merges two sorted containers and stores in new container in sorted order (merge sort). It takes 5 arguments, first and last iterator of 1st container, first and last iterator of 2nd container and 1st iterator of resultant container

Syntax:

```
	merge(beg1, end1, beg2, end2, beg3);
```

Program:

```
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
int main()
{
	vector<int> v1 = {1,2,3,4,5,6,7};
	vector<int> v2 = {10,20,30,40,50};
	vector<int> v3(12);
	merge(v1.begin(), v1.end(), v2.begin(),v2.end(), v3.begin());
	cout << "After merging :\n";
	for (int &x : v3)
		cout << x << " ";
	cout << endl;
	return 0;
}
```

6. inplace_merge() - This function is used to sort two consecutively placed sorted ranges in a single container. It takes 3 arguments, iterator to beginning of 1st sorted range, iterator to beginning of 2nd sorted range, and iterator to last position.

Syntax:

```
	inplace_merge(beg1, beg2, end);
```

Program:

```
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
int main()
{
	vector<int> v1 = {1,3,5,7,9};
	vector<int> v2 = {2,4,6,8,10};
	vector<int> v3(10);
	auto it = copy(v1.begin(), v1.end(), v3.begin());
	copy(v2.begin(), v2.end(), it);
	inplace_merge(v3.begin(),it,v3.end());
	cout << "After inplace_merging :\n";
	for (int &x : v3)
		cout << x << " ";
	cout << endl;
return 0;
}
```

Binary Search algorithms

Sl No	Operations	Description
1	lower_bound( )	Finds the first occurrence of a specified value
2	upper_bound( )	Finds the last occurrence of a specified value
3	binary_search( )	Conducts a binary search on an ordered sequence
4	equal_range( )	Finds a subrange of elements with a given value

1. lower_bound( ) - Returns an iterator pointing to the first element in the range [first, last) that does not satisfy element < value

Syntax:

```
ForwardIterator lower_bound (
     ForwardIterator first,
     ForwardIterator last,
     const T& val);
```

Program:

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

using namespace std;

int main()
{

  vector<int> v = {3, 1, 4, 6, 5};
  decltype(v)::iterator it = lower_bound(v.begin(), v.end(), 4);
  cout << "*it << ", pos = " << (it - v.begin()) << endl;
  return 0;

}
```

2. upper_bound( ) - It returns an iterator pointing to the first element in the range [first, last) that is greater than value, or last if no such element is found.

Syntax:

```
ForwardIterator upper_bound (
     ForwardIterator first,
     ForwardIterator last,
     const T& val);
```

Program:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
     vector<int> v{ 10, 20, 30, 40, 50 };
     cout << "Vector contains :";
     for (int i = 0; i < v.size(); i++)
          cout << " " << v[i];
     cout << "\n";
     vector<int>::iterator upper1, upper2;
     upper1 = upper_bound(v.begin(), v.end(), 35);
     upper2 = upper_bound(v.begin(), v.end(), 45);
     cout << "\nupper_bound for element 35 is at position : "
               << (upper1 - v.begin());
     cout << "\nupper_bound for element 45 is at position : "
               << (upper2 - v.begin());
     return 0;
}
```

3. binary_search( ) - It requires the array to be sorted before search is applied. It works by comparing the middle item of the array with our target, if it matches, it returns true otherwise if the middle term is greater than the target, the search is performed in the left sub-array.

Syntax:

```
binary_search(
     startaddress,
     endaddress,
    valuetofind)
```

Program:

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

void show(int a[], int size)
{
	for (int i = 0; i < size; ++i)
		cout << a[i] << ",";
}

int main()
{
	int a[] = { 1, 5, 8, 9, 6, 7, 3, 4, 2, 0 };
	int asize = sizeof(a) / sizeof(a[0]);
	cout << "\nThe array is : \n";
	show(a, asize);
	sort(a, a + asize);
	cout << "\n\nThe array after sorting  : \n";
	show(a, asize);
	if (binary_search(a, a + 10, 5))
		cout << "\nElement found in the array";
	else
		cout << "\nElement not found in the array";
	return 0;
}
```

4. equal_range( ) - It is used to find the sub-range within a given range [first, last) that has all the elements equivalent to a given value. It returns the initial and the final bound of such a sub-range.

Syntax:

```
pair equal_range (
            ForwardIterator first,
            ForwardIterator last,
                    const T& val);
```

Program:

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

int main()
{
        vector<int> v = { 1,1,2,2,2,3,1,3,3,7,7,8};
        pair<vector<int>::iterator, vector<int>::iterator> ip;

        sort(v.begin(), v.end());

        ip = equal_range(v.begin(), v.begin() + 12, 3);
        cout << "3 is present in the sorted vector from index " << (ip.first - v.begin())
        << " till "<< (ip.second - v.begin());
        return 0;
}
```

Heap algorithms

Sl No	Operations	Description
1	is_heap	checks if the given range is a max heap
2	is_heap_until	finds the largest subrange that is a max heap
3	make_heap( )	Makes a heap from a sequence
4	push_heap( )	Adds an element to heap
5	pop_heap( )	Deletes the top element
6	sort_heap( )	Sorts a heap

1. is_heap( ) - It is used to check whether a given range of elements forms Max Heap or not. It returns True when given ranges of elements forms Max Heap, else it returns False.

Syntax:

```
    is_heap(first, last)
```

Program:

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

using namespace std;

bool ignore_case(char a, char b) {
  return (tolower(a) == tolower(b));
}

int main(void) {
  vector<char> v = {'E', 'd', 'C', 'b', 'A'};
  bool result;

  result = is_heap(v.begin(), v.end());

  if (result == false)
```

```
      cout << "Given sequence is not a max heap." << endl;

   result = is_heap(v.begin(), v.end(), ignore_case);

   if (result == true)
      cout << "Given sequence is a max heap." << endl;
}
```

2. is_heap_until( ) - It is used to return an iterator pointing at the first element in the range [first, last) that does not satisfy the heap ordering condition, or end if the range forms a heap.

Syntax:

```
is_heap_until (
               RandomAccessIterator first,
               RandomAccessIterator last);
```

Program:

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

using namespace std;

int main () {
  vector<int> arr {2,6,9,3,8,4,5,1,7};

  sort(arr.begin(),arr.end());
  reverse(arr.begin(),arr.end());

  auto last = is_heap_until (arr.begin(),arr.end());

  cout << "The " << (last-arr.begin()) << " first elements are a valid heap:";
  for (auto it=arr.begin(); it!=last; ++it)
    cout << ' ' << *it;
  cout << '\n';

  return 0;
}
```

3. make_heap () - It is used to rearrange the elements in the range [first, last) in such a way that they form a heap.

Syntax:

```
 make_heap (
     RandomAccessIterator first,
     RandomAccessIterator last);
```

Program:

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

int main()
{
  vector<int> v = {2, 1, 4, 3};

  make_heap(v.begin(), v.end());

  for_each(v.begin(), v.end(), [](int x)
 {
    cout << x << endl;
  });

  return 0;
}
```

4. push_heap( ) - This function is used to insert elements into heap. The size of the heap is increased by 1. New element is placed appropriately in the heap.

Program:

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

using namespace std;
void print(const vector <int>& v)
{
    vector <int>::const_iterator i;
    for (i = v.begin(); i != v.end(); i++)
        cout << *i << " ";
    cout << endl;
}

int main()
{
    int arr[] = {1, 8, 3, 2, 4, 6};
    vector <int> v(arr, arr + sizeof(arr) / sizeof(int));
    cout << "vector v : ";
    print(v);

    make_heap(v.begin(), v.end());
    cout << "Heap : ";
    print(v);



    v.push_back(7);
    push_heap(v.begin(), v.end());
    cout << "Heap after push : ";
    print(v);
}
```

5. pop_heap( ) - This function is used to delete the maximum element of the heap. The size of heap is decreased by 1. The heap elements are reorganised accordingly after this operation.

Program:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{

        vector<int> v1 = {20, 30, 40, 25, 15};
        make_heap(v1.begin(), v1.end());
        cout << "The maximum element of heap is : ";
        cout << v1.front() << endl;
        // using pop_heap() to delete maximum element
        pop_heap(v1.begin(), v1.end());
        v1.pop_back();
        // Displaying the maximum element of heap
        // using front()
        cout << "The maximum element of heap after pop is : ";
        cout << v1.front() << endl;

        return 0;
}
```

6. sort_heap( ) - This function is used to sort the heap. After this operation, the container is no longer a heap.

Program:

```
#include <iostream>
```

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

int main()
{
	vector<int> v1 = {20, 30, 40, 25, 15};
	make_heap(v1.begin(), v1.end());
	cout << "The heap elements are : ";
	for (int &x : v1)
	cout << x << " ";
	cout << endl;
	 is_heap_until (arr.begin(),arr.end());
	 push_heap(v.begin(), v.end());
	pop_heap(v1.begin(), v1.end());
	// sorting heap using sort_heap()
	sort_heap(v1.begin(), v1.end());

	// Displaying heap elements
	cout << "The heap elements are : ";
	for (int &x : v1)
	cout << x << " ";
	cout << endl;

	return 0;
}
```

Partition algorithms

Sl No	Operations	Description
1	is_partitioned	Determines if the range is partitioned by the given predicate
2	Partition( )	Places elements matching a predicate first
3	partition_copy	copies a range dividing the elements into two groups
4	stable_partition	Divides elements into two groups while preserving their relative order
5	partition_point	It locates the partition point of a partitioned range

1. is_partitioned( ) - It is used to test to see if a range [first, last) is partitioned according to a predicate. In other words, all the elements in the range that satisfies the predicate are at the beginning of the sequence.

Syntax:

```
  bool is_partitioned (
     InputIterator first,
     InputIterator last,
     UnaryPredicate pred);
```

Program:

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

bool pred(int a)
{
	return (a % 2 == 0);
}

using namespace std;


int main()
{

	vector<int> v1 = { 2, 4, 6, 3, 5, 7, 9 };


	bool b = is_partitioned(v1.begin(), v1.end(), pred);

	if (b == 1) {
		cout << "All the even no. are present before odd no.";
	} else {
		cout << "All the even no. are not present before odd no.";
	}

	v1.push_back(16);
```

```
        b = is_partitioned(v1.begin(), v1.end(), pred);

        if (b == 1)
                cout << "\nAll the even no. are present before odd no.";
        else
                cout << "\nAll the even no. are not present before odd no.";


        return 0;
}
```

2. partition( ) - This function is used to partition the elements on basis of condition mentioned in its arguments.

Syntax:

```
    partition (
        ForwardIterator first,
        ForwardIterator last,
        UnaryPredicate pred);
```

Program:

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

using namespace std;

bool IsOdd (int i) { return (i%2)==1; }

int main () {
  vector<int> myvector;
  for (int i=1; i<10; ++i)
        myvector.push_back(i);

  vector<int>::iterator bound;
  bound = partition (myvector.begin(), myvector.end(), IsOdd);

  // print out content:
  cout << "odd elements:";
  for (vector<int>::iterator it=myvector.begin(); it!=bound; ++it)
```

```
    cout << ' ' << *it;
  cout << '\n';

  cout << "even elements:";
  for (vector<int>::iterator it=bound; it!=myvector.end(); ++it)
    cout << ' ' << *it;
  cout << '\n';

  return 0;
}
```

3. partition_copy ( ) - This function is used to partition the elements on basis of condition mentioned in its arguments.

Syntax:

```
 partition_copy (
     InputIterator first,
     InputIterator last,
     OutputIterator1 result_true,
     OutputIterator2 result_false,
     UnaryPredicate pred);
```

Program:

```
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
bool mul3(int x)
{
return (x % 3);
}
int main()
{
vector<int> vect = { 12, 15, 9, 7, 84, 74 };
vector<int> vec1;
vector<int> vec2;
int x = count_if (vect.begin(), vect.end(), mul3);
vec1.resize(x);
vec2.resize(vect.size()-x);

partition_copy(vect.begin(), vect.end(), vec1.begin(),vec2.begin(), mul3);
cout << "The elements that return true for condition i.e., function are : ";
for (int &x : vec1)
cout << x << " ";
cout << endl;

cout << "The elements that return false for condition i.e., function are : ";
for (int &x : vec2)
cout << x << " ";
cout << endl;

return 0;
}
```

4. stable_partition( ) - This function is used to partition the elements on basis of condition mentioned in its arguments in such a way that the relative order of the elements is preserved.

Syntax:

```
    stable_partition(beg, end, condition)
```

Program:

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

int main()
{
        vector<int> vect = { 2, 1, 5, 6, 8, 7 };
        stable_partition(vect.begin(), vect.end(), [](int x)
        {
                return x%2 == 0;
        });

        cout << "The partitioned vector is : ";
        for (int &x : vect) cout << x << " ";
        cout << endl;
        return 0;

}
```

5. partition_point( ) - This function returns an iterator pointing to the partition point of container i.e. the first element in the partitioned range [beg,end) for which condition is not true. The container should already be partitioned for this function to work.

Syntax:

```
    partition_point(beg, end, condition)
```

Program:

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

bool mul3(int x){
    return (x % 3);
}

int main(){
    vector<int> vect = { 12, 15, 9, 7, 84, 74 };

    partition(vect.begin(), vect.end(), mul3);
    cout << "The partitioned vector is : ";
    for (int &x : vect)
        cout << x << " ";
    cout << endl;

    vector<int>::iterator i;
    auto it = partition_point(vect.begin(), vect.end(), mul3);
    cout << "The partition which satisfies the condition is: ";
    for ( i= vect.begin(); i!=it; i++)
        cout << *i << " ";
    cout << endl;
}
```

Set algorithms

Sl No	Operations	Description
1	includes( )	Finds whether a sequence is a subsequence of another
2	set_difference( )	Constructs a sequence that is the difference of two ordered sets
3	set_intersection( )	Constructs a sequence that contains the intersection of ordered sets
4	set_symmetric_difference()	Produces a set which is the symmetric difference between two ordered sets
5	set_union( )	Produces sorted union of two ordered sets

1. includes( ) - This function is used to recognize if all the numbers in a container, also exist in other containers. It helps to check whether a set is a subset of another set or not considering the set is ordered

Syntax:

```
bool includes (initer1 beg1, initer1 end1,
                    initer2 beg2, initer2 end2,)
```

Program:

```
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
int main()
{
        vector<int> arr1 = { 1, 5, 6, 3, 2, 4 };
        vector<int> arr2 = { 1, 5, 3, 4 };

        sort(arr1.begin(), arr1.end());
        sort(arr2.begin(), arr2.end());
if (includes(arr1.begin(), arr1.end(), arr2.begin(), arr2.end()))
cout << "All elements of 2nd container are in 1st container";
else
cout << "All elements of 2nd container are not in 1st container";
return 0;
}
```

2. set_difference( ) - The difference of two sets is formed by the elements that are present in the first set, but not in the second one. The elements copied by the function come always from the first range, in the same order. The elements in the both the ranges shall already be ordered.

Syntax:

```
    OutputIterator set_difference (
                   InputIterator1 first1,
                   InputIterator1 last1,
                   InputIterator2 first2,
                   InputIterator2 last2,
                   OutputIterator result);
```

Program:

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

int main()
{
        int first[] = { 5, 10, 15, 20, 25 };
        int second[] = { 50, 40, 30, 20, 10 };
        int n = sizeof(first) / sizeof(first[0]);
        vector<int> v2(5);
        vector<int>::iterator it, ls;

        sort(first, first + 5);
        sort(second, second + 5);

        // Print elements
        cout << "First array :";
        for (int i = 0; i < n; i++)
                cout << " " << first[i];
        cout << "\n";

        // Print elements
        cout << "Second array :";
        for (int i = 0; i < n; i++)
                cout << " " << second[i];
        cout << "\n\n";


        ls = set_difference(first, first + 5, second, second + 5, v2.begin());

        cout << "Using default comparison, \n";
        cout << "The difference has " << (ls - v2.begin()) << " elements :";
        for (it = v2.begin(); it < ls; ++it)
                cout << " " << *it;
        cout << "\n";

        return 0;
}
```

3. set_intersection( ) - The intersection of two sets is formed only by the elements that are present in both sets. The elements copied by the function come always from the first range, in the same order. The elements in the both the ranges shall already be ordered.

Syntax:

```
    set_intersection (
        InputIterator1 first1,
        InputIterator1 last1,
        InputIterator2 first2,
        InputIterator2 last2,
        OutputIterator result);
```

Program:

```
#include<iostream>
```

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

int main()
{
        int first[] = { 5, 10, 15, 20, 25 };
        int second[] = { 50, 40, 30, 20, 10 };
        int n = sizeof(first) / sizeof(first[0]);

        vector<int> v1(5);
        vector<int> v2(5);
        vector<int>::iterator it, ls;

        sort(first, first + 5);
        sort(second, second + 5);

        cout << "First array :";
        for (int i = 0; i < n; i++)
                cout << " " << first[i];
        cout << "\n";



        cout << "Second array :";
        for (int i = 0; i < n; i++)
                cout << " " << second[i];
        cout << "\n\n";

        ls = set_intersection(first, first + 5, second, second + 5, v1.begin());

        cout << "The intersection has " << (ls - v1.begin()) << " elements:";
        for (it = v1.begin(); it != ls; ++it)
                cout << ' ' << *it;
        cout << "\n";

        return 0;
}
```

4. set_symmetric_difference( ) - The symmetric difference between two sets is formed by the elements that are present in one of the sets, but not in the other. Among the equivalent elements in each range, those discarded are those that appear before in the existent order before the call. The existing order is also preserved for the copied elements.

Syntax:

```
set_symmetric_difference (
InputIterator1 first1,
InputIterator1 last1,
InputIterator2 first2,
InputIterator2 last2,
OutputIterator result);
```

Program:

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

int main()
{
        int first[] = { 5, 10, 15, 20, 25 };
        int second[] = { 50, 40, 30, 20, 10 };
        int n = sizeof(first) / sizeof(first[0]);

        cout << "First array contains :";
        for (int i = 0; i < n; i++)
                cout << " " << first[i];
        cout << "\n";

        cout << "Second array contains :";
        for (int i = 0; i < n; i++)
                cout << " " << second[i];
        cout << "\n\n";

        vector<int> v(10);
        vector<int>::iterator it, st;
        sort(first, first + 5);
        sort(second, second + 5);

        it = set_symmetric_difference(first, first + 5,
        second, second + 5, v.begin());

        cout << "The symmetric difference has "
                        << (it - v.begin()) << " elements:\n";
        for (st = v.begin(); st != it; ++st)
        cout << ' ' << *st;
        cout << '\n';

        return 0;
}
```

5. set_union( ) - The union of two sets is formed by the elements that are present in either one of the sets, or in both. Elements from the second range that have an equivalent element in the first range are not copied to the resulting range.

Syntax:

```
        set_union (
                InputIterator1 first1,
                InputIterator1 last1,
                InputIterator2 first2,
                InputIterator2 last2,
                OutputIterator result);
```

Program:

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

int main()
{
        int first[] = { 5, 10, 15, 20, 25 };
        int second[] = { 50, 40, 30, 20, 10 };
        int n = sizeof(first) / sizeof(first[0]);
        cout << "First array contains :";
        for (int i = 0; i < n; i++)
                cout << " " << first[i];
        cout << "\n";

        cout << "Second array contains :";
        for (int i = 0; i < n; i++)
                cout << " " << second[i];
        cout << "\n\n";

        vector<int> v(10);
        vector<int>::iterator it, st;

        sort(first, first + n);
        sort(second, second + n);

        it = set_union(first, first + n, second, second + n, v.begin());

        cout << "The union has " << (it - v.begin())
                        << " elements:\n";
        for (st = v.begin(); st != it; ++st)
                cout << ' ' << *st;
        cout << '\n';

        return 0;
}
```

Relational algorithms

Sl No	**Operations**	**Description**
1	equal( )	Finds whether two sequences are the same
2	lexicographical_compare()	Compares alphabetically one sequence with other
3	max( )	Gives maximum of two values
4	max_element( )	Finds the maximum element within a sequence
5	min( )	Gives minimum of two values
6	min_element( )	Finds the minimum element within a sequence
7	mismatch()	Finds the first mismatch between the

		elements in two sequences

1. equal( ) - It tests whether two sets of element are equal or not. Size of the both set need not to be equal. It uses binary predicate for comparison.

Syntax:

```
          equal(
               InputIterator1 first1,
               InputIterator1 last1,
               InputIterator2 first2,
               BinaryPredicate pred);
```

Program:

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

using namespace std;

/* Binary predicate which always returns true */
bool binary_pred(string s1, string s2) {
   return true;
}

int main(void) {
   vector<string> v1 = {"one", "two", "three"};
   vector<string> v2 = {"ONE", "THREE", "THREE"};
   bool result;

   result = equal(v1.begin(), v1.end(), v2.begin(), binary_pred);

   if (result == true)
      cout << "Vector range is equal." << endl;

   return 0;
}
```

2. lexicographical_compare( ) - It checks whether one range is lexicographically less than another or not. A lexicographical comparison is the kind of comparison generally used to sort words alphabetically in dictionaries

Syntax:

```
lexicographical_compare(
     InputIterator1 first1,
     InputIterator1 last1,
     InputIterator2 first2,
     InputIterator2 last2);
```

Program:

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

using namespace std;

int main(void) {
  vector<string> v1 = {"One", "Two", "Three"};
  vector<string> v2 = {"one", "two", "three"};
  bool result;

  result = lexicographical_compare(v1.begin(), v1.end(), v2.begin(), v2.end());

  if (result == true)
    cout << "v1 is less than v2." << endl;

  v1[0] = "two";

  result = lexicographical_compare(v1.begin(), v1.end(), v2.begin(), v2.end());

  if (result == false)
    cout << "v1 is not less than v2." << endl;

  return 0;
}
```

3. max( ) - It is used to find out the largest of the number passed to it. It returns the first of them, if there are more than one.

Syntax:

```
max (const T& a, const T& b);
```

Program:

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

int main()
{
    cout << "max(10,20)       : " << max(10, 20) << endl;
    cout << "max(10.23f,20.12f): " << max(10.23f, 20.12f) << endl;
    cout << "max(-10,-20)      : " << max(-10, -20) << endl;
    cout << "max('A','a')      : " << max('A', 'a') << endl;
    cout << "max('A','Z')      : " << max('A', 'Z') << endl;
    cout << "max(10,10)        : " << max(10, 10) << endl;

    return 0;
}
```

2. max_element( ) - It returns an iterator pointing to the element with the largest value in the range [first, last).

Syntax:

```
max_element (
     ForwardIterator first,
     ForwardIterator last);
```

Program:

```
#include <iostream>
#include <algorithm>
using namespace std;
int main()
{
int v[] = { 'a', 'c', 'k', 'd', 'e', 'f', 'h' };                //k

int* i1;
i1 = max_element(v, v + 4);

cout << char(*i1) << "\n";
return 0;
}
```

2. min( ) - is used to find out the smallest of the number passed to it. It returns the first of them, if there are more than one.

Syntax:

```
          template  constexpr const T& min (
                    const T& a,
                    const T& b);
```

Program:

```
#include <iostream>
#include <algorithm>
using namespace std;
int main()
{
        int a = 5;
        int b = 7;
        cout << min(a, b) << "\n";
        return 0;
}
```

2. min_element( ) - It returns an iterator pointing to the element with the smallest value in the range [first, last).

Syntax:

```
        min_element (
            ForwardIterator first,
            ForwardIterator last);
```

Program:

```
#include <iostream>
#include <algorithm>
using namespace std;
int main()
{
    int v[] = { 9, 4, 7, 2, 5, 10, 11, 12, 1, 3, 6 };
    int* i1;
    i1 = min_element(v + 2, v + 5);
    cout << *i1 << "\n";
    return 0;
}
```

Numeric algorithms

Sl No	Operations	Description
1	accumulate( )	Accumulates the results of operation on a sequence
2	adjacent_difference()	Produces a sequence from another sequence
3	inner_product( )	Accumulates the results of operation on a pair of sequences
4	partial_sum( )	Produces a sequence by operation on a pair of sequences

1. accumulate( ) -

Syntax:

```
        min_element (
            ForwardIterator first,
            ForwardIterator last);
```

Program:

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

int main() {
        vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        int sum = accumulate(v.begin(), v.end(), 0);
        cout << sum;
}
```

2. adjacent_difference() - Assigns to every element in the range starting at result, the difference between its corresponding element in the range [first, last] and the one preceding it (except for *result, which is assigned *first).

Syntax:

```
        adjacent_difference (
                        InputIterator first,
                        InputIteratorlast,
                        OutputIterator result);
```

Program:

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

int main()
{
        int val[] = { 1, 2, 3, 5, 9, 11, 12 };
        int n = sizeof(val) / sizeof(val[0]);
        int result[7];
        cout << "Array contains :";
        for (int i = 0; i < n; i++)
                cout << " " << val[i];
        cout << "\n";

        adjacent_difference(val, val + 7, result);
        cout << "Using default adjacent_difference: ";
        for (int i = 1; i < n; i++)
                cout << result[i] << ' ';
        cout << '\n';
        return 0;
}
```

3. inner_product( )- Returns the result of accumulating init with the inner products of the pairs formed by the elements of two ranges starting at first1 and first2.
The two default operations (to add up the result of multiplying the pairs) may be overridden by the arguments binary_op1 and binary_op2.

Syntax:

```
        inner_product (
                InputIterator1 first1,
                InputIterator1 last1,
                InputIterator2 first2,
                T init);
```

Program:

```
#include <iostream>
#include <functional>
#include <numeric>

int main()
{
        int init = 100;
        int series1[] = { 10, 20, 30 };
        int series2[] = { 1, 2, 3 };
        int n = sizeof(series1) / sizeof(series1[0]);
        cout << "First array contains :";
        for (int i = 0; i < n; i++)
                cout << " " << series1[i];
        cout << "\n";
        cout << "Second array contains :";
        for (int i = 0; i < n; i++)
                cout << " " << series2[i];
        cout << "\n\n";
        cout << "Using default inner_product: ";
        cout << inner_product(series1, series1 + n, series2, init);
        cout << '\n';

        return 0;
}
```

4. partial_sum( )- This function assigns a partial sum of the corresponding elements of an array to every position of the second array. It returns the partial sum of all the sets of values lying between [first, last) and stores it in another array b.

Syntax:

```
	partial_sum(first, last, b);
```

Program:

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

int myfun(int x, int y)
{
	return x + 2 * y;
}

int main()
{
	int a[] = { 1, 2, 3, 4, 5 };
	int b[5];

	partial_sum(a, a + 5, b);
	cout << "Partial Sum - Using Default function: ";
	for (int i = 0; i < 5; i++)
		cout << b[i] << ' ';
	cout << '\n';

	partial_sum(a, a + 5, b, myfun);
	cout << "Partial sum - Using user defined function: ";
	for (int i = 0; i < 5; i++)
		cout << b[i] << ' ';
	cout << '\n';

	return 0;
}
```

FUNCTION OBJECTS (Functors)

A Function object is a function wrapped in a class so that it looks like an object. A function object extends the characteristics of a regular function by using the feature of an object oriented such as generic programming. Therefore, we can say that the function object is a smart pointer that has many advantages over the normal function. <functional.h>

Advantages of function objects

- Function objects can have member functions as well as member attributes.
- Function objects can be initialized before their usage.
- Regular functions can have different types only when the signature differs. Function objects can have different types even when the signature is the same.
- Function objects are faster than the regular function.

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

class Student
{
    public:
        void operator()()
        {
            cout<<"Function object";
        }
};

int main()
{
    Student s;
    s();

    return 0;

}
```

A function object is also known as a 'functor'. A function object is an object that contains atleast one definition of operator() function. It means that if we declare the object 'd' of a class in which operator() function is defined, we can use the object 'd' as a regular function

Function objects are often used as arguments to certain containers and algorithm.

For example, the statement

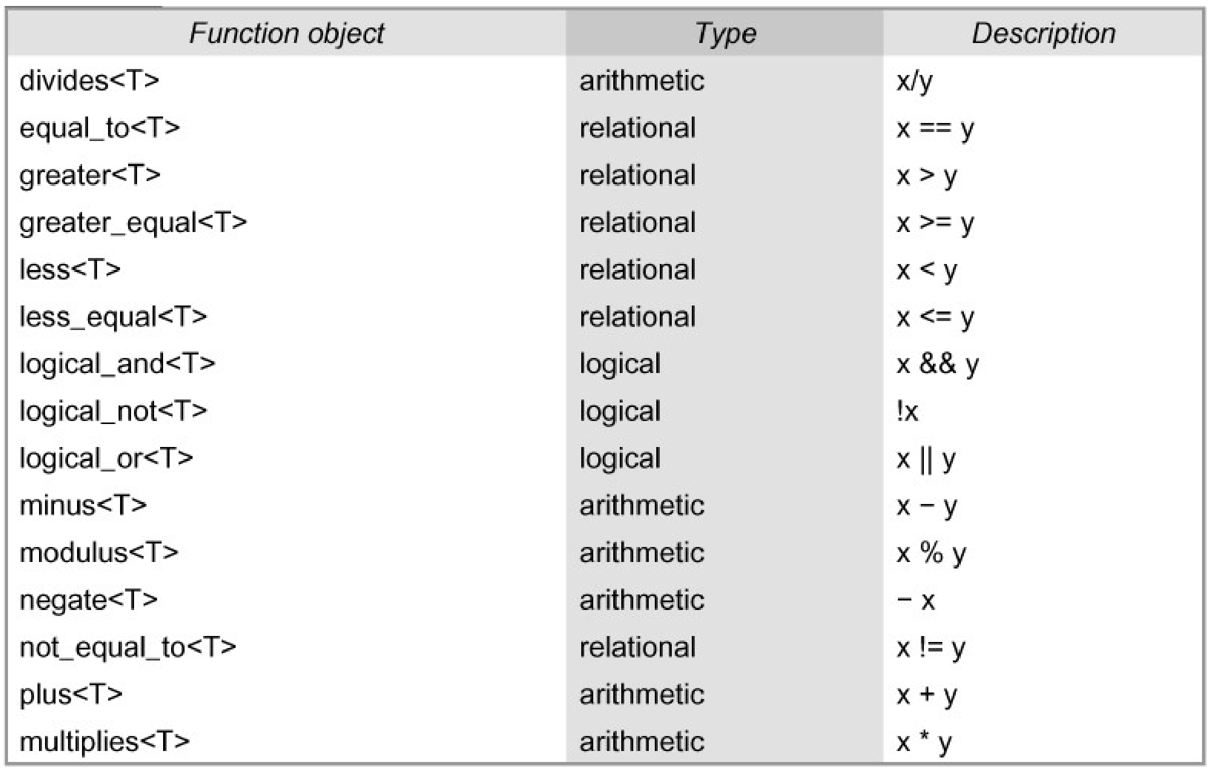
```
         sort(array, array+5, greater<int>());
```

uses the function object greater<int>( ) to sort the elements contained in array in descending order.

```
#include <iostream>
#include<algorithm>
#include<functional>
using namespace std;

int main()
{
    int a[5]={1,2,3,4,5};
    sort(a,a+5,less<int>());
    for(int i=0;i<5;i++)
    {
        cout<<a[i]<<" ";
    }

    return 0;
}
```

STL provides many other predefined function objects for performing arithmetical and logical operations, For using function objects, we must include <functional> header file.

STL function objects in <functional>



Function object	*Type*	*Description*
divides<T>	arithmetic	x/y
equal_to<T>	relational	x == y
greater<T>	relational	x > y
greater_equal<T>	relational	x >= y
less<T>	relational	x < y
less_equal<T>	relational	x <= y
logical_and<T>	logical	x && y
logical_not<T>	logical	!x
logical_or<T>	logical	x || y
minus<T>	arithmetic	x − y
modulus<T>	arithmetic	x % y
negate<T>	arithmetic	− x
not_equal_to<T>	relational	x != y
plus<T>	arithmetic	x + y
multiplies<T>	arithmetic	x * y

Note: The variables x and y represent objects of class T passed to the function object as arguments.