<u>Unit -V</u>

Templates:

Introduction to Templates, Class Templates, Class Templates with Multiple Parameters, Function Templates, Function Templates with Multiple Parameters.
Standard Template Library Classes: STL Container classes-Array class, Vector,stack,queue, STL Algorithm classes- Sort, reverse, max, min.
Application Development using C++.

<u>Introduction to Templates:</u>

The simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types(Ex:Function Overloading). For example, a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

So,we need to write the program with function (or) class to support any type of data.Such programming is called "Generic Programming".

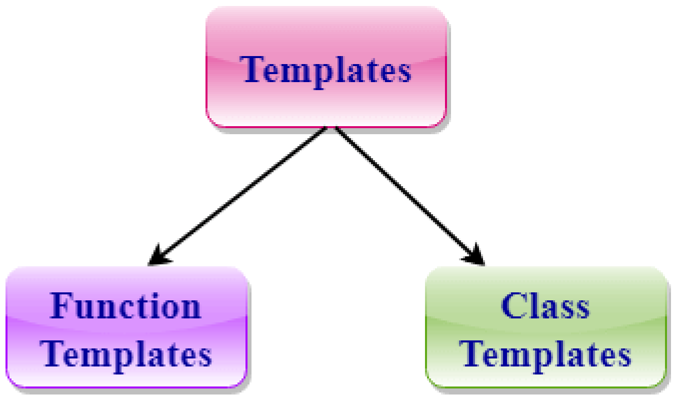
To implement Generic programming we can use the "Templates". C++ adds two new keywords to support templates: *'template'* and *'typename'*. The second keyword can always be replaced by keyword '*class*'.

<u>C++ Templates</u>

A C++ template is a powerful feature added to C++. It allows you to define the generic classes and generic functions and thus provides support for generic programming. Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

Templates can be represented in two ways:

- Function templates
- Class templates



Function Template

- Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.
- The type of the data that the function will operate on depends on the type of the data passed as a parameter.
- For example, Quick sorting algorithm is implemented using a generic function, it can be implemented to an array of integers or array of floats.
- A Generic function is created by using the keyword template. The template defines what function will do.

Syntax of Function Template

```
template < class Ttype>
ret_type func_name(parameter_list)
        {
           // body of function.
        }
```

Where **Ttype**: It is a placeholder name for a data type used by the function. It is used within the function definition. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.

class: A class keyword is used to specify a generic type in a template declaration.

Example:

```
#include <iostream>
using namespace std;
template<class T>
T Addition(T  a,T  b)
{
   T result = a+b;
   return result;

}
int main()
{
           Cout<<"Sumof Integers="<<Addition(10,20);
          Cout<<"Sum of floaalues="<<Additon(10.5,25.5);
}
```

Function Templates with Multiple Parameters

We can use more than one generic type in the template function by using the comma to separate the list.

Syntax:

```
template<class T1, class T2,.....>
return_type function_name (arguments of type T1, T2....)
{
   // body of function.
}
```

In the above syntax, we have seen that the template function can accept any number of arguments of a different type.

Example:

```
            #include <iostream>
            using namespace std;
            template<class T1,class T2>
            T1 fun(T1  a, T2  b)
            {
                Cout<<"Addition="<<a+b;
                //cout << "Value of a is : " <<a<< endl;
                //cout << "Value of b is : " <<b<< endl;
            }
 ).
          int main()
          {
                fun(15,12.3);

               return 0;
          }
```

Class Template

Class Template can also be defined similarly to the Function Template. When a class uses the concept of Template, then the class is known as generic class.

Syntax

```
template<class Ttype>
class class_name
{
  .
  .
}
```

Ttype is a placeholder name which will be determined when the class is instantiated.

We can define more than one generic data type using a comma-separated list. The Ttype can be used inside the class body.

Now, we create an instance of a class

```
class_name<type> ob;
```

where class_name: It is the name of the class.

type: It is the type of the data that the class is operating on.

ob: It is the name of the object.

```
#include <iostream>
using namespace std;
template<class T>
class A
{
    public:
    T num1 = 5;
    T num2 = 6;
    void add()
    {
         cout << "Addition of num1 and num2 :  << num1+num2<<endl;
     }

 };

int main()
{
     A<int> d;
     d.add();
     return 0;
}
```

Class Template with Multiple Parameters

We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

Syntax:

```
template<class T1, class T2, ......>

class class_name
{
  // Body of the class.
}
```

Example:

```
#include <iostream>
 using namespace std;
 template<class T1, class T2>
class A
{
    T1 a;
    T2 b;
    public:
   A(T1 x,T2 y)
  {
      a = x;
      b = y;
   }
      void display()
     {
           cout << "Values of a and b are : " << a<<" ,"<<b<<endl;
      }
 };

 int main()
 {
      A<int,float> d(5,6.5);
      d.display();
      return 0;
 }
```

Nontype Template Arguments

The template can contain multiple arguments, and we can also use the non-type arguments in addition to the type T argument, we can also use other types of arguments such as strings, function names, constant expression and built-in types.

Example:

```
template<class T, int size>
class array
{
        T arr[size];           // automatic array initialization.
};
```

In the above case, the nontype template argument is size and therefore, template supplies the size of the array as an argument.

Arguments are specified when the objects of a class are created:

```
array<int, 15> t1;                    // array of 15 integers.
array<float, 10> t2;                  // array of 10 floats.
array<char, 4> t3;                    // array of 4 chars.
```

Example:

```cpp
#include <iostream>
using namespace std;
template<class T, int size>
class A
{
   public:
   T arr[size];
   void insert()
   {
      int i =1;
      for (int j=0;j<size;j++)
      {
         arr[j] = i;
         i++;
      }
   }

   void display()
   {
      for(int i=0;i<size;i++)
      {
         std::cout << arr[i] << " ";
      }
   }
};
int main()
{
   A<int,10> t1;
   t1.insert();
   t1.display();
   return 0;
}
```

Points to Remember

- C++ supports a powerful feature known as a template to implement the concept of generic programming.
- A template allows us to create a family of classes or family of functions to handle different data types.
- Template classes and functions eliminate the code duplication of different data types and thus makes the development easier and faster.
- Multiple parameters can be used in both class and function template.
- Template functions can also be overloaded.
- We can also use nontype arguments such as built-in or derived data types as template arguments.

Standard Template Library Classes(STL)

Introduction to STL: Standard Template Library

- STL is an acronym for standard template library.
- It is a set of C++ template classes that provide generic classes and function that can be used to implement data structures and algorithms .

STL is mainly composed of 3 Components:

1. Containers
2. Algorithms
3. Iterators

C++: Containers in STL

- Container library in STL provide containers that are used to create data structures like arrays, linked list, trees etc.
- These container are generic, they can hold elements of any data types.
- Example: **vector** can be used for creating dynamic arrays of char, integer, float and other types.

C++: Algorithms in STL

- STL provide number of algorithms that can be used of any container, irrespective of their type.
- Algorithms library contains built in functions that performs complex algorithms on the data structures.
- Example: one can reverse a range with reverse() function, sort a range with sort() function, search in a range with binary_search() and so on.
- Algorithm library provides abstraction, i.e you don't necessarily need to know how the the algorithm(function) works.

C++: Iterators in STL

- Iterators in STL are used to point to the containers.
- Iterators actually acts as a bridge between containers and algorithms.
- Example: sort() algorithm have two parameters, starting iterator and ending iterator, now sort() compare the elements pointed by each of these iterators and arrange them in sorted order, thus it does not matter what is the type of the container and same sort() can be used on different types of containers.

<u>Use and Application of STL</u>

- STL being generic library provide containers and algorithms which can be used to store and manipulate different types of data thus it saves us from defining these data structures and algorithms from the scratch.
- Because of STL, now we do not have to define our sort function every time we make a new program or define same function twice for the different data types, instead we can just use the generic container and algorithms in STL.
- This saves a lot of time, code and effort during programming, thus STL is heavily used in the competitive programming, plus it is reliable and fast.

STL Container Classes

1.Array(Sequential Container)

2.Vector(Sequential Container)

3.Stack(Container Adaptor)

4.Queue(Container Adaptor)

Array class in C++

The introduction of array class from C++ has offered a better alternative for C-style arrays. The advantages of array class over C-style array are :-

- Array classes knows its own size, whereas C-style arrays lack this property. So when passing to functions, we don't need to pass size of Array as a separate parameter.
- With C-style array there is more risk of array being decayed into a pointer. Array classes don't decay into pointers.
- Array classes are generally more efficient, light-weight and reliable than C-style arrays.

Operations on Array :-

1. at() :- This function is used to access the elements of array from specified position.

2. operator[] :- This is similar to C-style arrays. This method is also used to access array elements.

4. front() :- This returns the first element of array.
5. back() :- This returns the last element of array.

6. size() :- It returns the number of elements in array. This is a property that C-style arrays lack.
7. max_size() :- It returns the maximum number of elements array can hold i.e, the size with which array is declared. The size() and max_size() return the same value.

8. swap() :- The swap() swaps all elements of one array with other.

9. empty() :- This function returns true when the array size is zero else returns false.
10. fill() :- This function is used to fill the entire array with a particular value.

// C++ code to demonstrate working of array functions at(),operator[]

```
#include<iostream>
#include<array> // for array, at()
using namespace std;
int main()
{
   // Initializing the array elements
   array<int,6> ar = {1, 2, 3, 4, 5, 6};

   // Printing array elements using at()
   cout << "The array elements are (using at()) : ";
   for ( int i=0; i<6; i++)
   cout << ar.at(i) << " ";
   cout << endl;


   // Printing array elements using operator[]
   cout << "The array elements are (using operator[]) : ";
   for ( int i=0; i<6; i++)
   cout << ar[i] << " ";
   cout << endl;

   return 0;

}
```

Output:

```
The array elemets are (using at()) : 1 2 3 4 5 6
The array elements are (using operator[]) : 1 2 3 4 5 6
```

// C++ code to demonstrate working of front() and back()

```
#include<iostream>
#include<array> // for front() and back()
using namespace std;
int main()
{
   // Initializing the array elements
   array<int,6> ar = {1, 2, 3, 4, 5, 6};

   // Printing first element of array
   cout << "First element of array is : ";
   cout << ar.front() << endl;

   // Printing last element of array
   cout << "Last element of array is : ";
   cout << ar.back() << endl;

   return 0;

}
```

<u>Output</u>:

```
First element of array is : 1
Last element of array is : 6
```

// C++ code to demonstrate working of size()and max_size()

```
#include<iostream>
#include<array> // for size() and max_size()
using namespace std;
int main()
{
   // Initializing the array elements
   array<int,6> ar = {1, 2, 3, 4, 5, 6};

   // Printing number of array elements
   cout << "The number of array elements is : ";
```

```
    cout << ar.size() << endl;

    // Printing maximum elements array can hold
    cout << "Maximum elements array can hold is : ";
    cout << ar.max_size() << endl;

    return 0;

}
```

Output:

```
The number of array elements is : 6
Maximum elements array can hold is : 6
```

// C++ code to demonstrate working of swap()

```
#include<iostream>
#include<array> // for swap() and array
using namespace std;
int main()
{

    // Initializing 1st array
    array<int,6> ar = {1, 2, 3, 4, 5, 6};

    // Initializing 2nd array
    array<int,6> ar1 = {7, 8, 9, 10, 11, 12};

    // Printing 1st and 2nd array before swapping
    cout << "The first array elements before swapping are : ";
    for (int i=0; i<6; i++)
    cout << ar[i] << " ";
    cout << endl;
    cout << "The second array elements before swapping are : ";
    for (int i=0; i<6; i++)
    cout << ar1[i] << " ";
    cout << endl;
    // Swapping ar1 values with ar
      ar.swap(ar1);
    // Printing 1st and 2nd array after swapping
    cout << "The first array elements after swapping are : ";
    for (int i=0; i<6; i++)
    cout << ar[i] << " ";
    cout << endl;
    cout << "The second array elements after swapping are : ";
    for (int i=0; i<6; i++)
    cout << ar1[i] << " ";
    cout << endl;
      return 0;

}
```

Output:

```
The first array elements before swapping are : 1 2 3 4 5 6
The second array elements before swapping are : 7 8 9 10 11 12
The first array elements after swapping are : 7 8 9 10 11 12
The second array elements after swapping are : 1 2 3 4 5 6
```

// C++ code to demonstrate working of empty() and fill()

```
#include<iostream>
#include<array> // for fill() and empty()
using namespace std;
int main()
{

    // Declaring 1st array
         array<int,6> ar;

    // Declaring 2nd array
         array<int,0> ar1;

    // Checking size of array if it is empty
     ar1.empty()? cout << "Array empty":cout << "Array not empty";
        cout << endl;

    // Filling array with 0
             ar.fill(0);

    // Displaying array after filling
```

```
        cout << "Array after filling operation is : ";
        for ( int i=0; i<6; i++)
            cout << ar[i] << " ";

        return 0;

    }
```

Output:

```
Array empty
Array after filling operation is : 0 0 0 0 0 0
```

Vector in C++ STL

Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container.

Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.

In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes there may be a need of extending the array.

Removing the last element takes only constant time because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.

Certain functions associated with the vector are:

Iterator:

begin() – Returns an iterator pointing to the first element in the vector
end() – Returns an iterator pointing to the theoretical element that follows the last element in the vector
rbegin() – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
rend() – Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

```
/*cbegin() – Returns a constant iterator pointing to the first element in the
vector.
cend() – Returns a constant iterator pointing to the theoretical element that
          follows the last element in the vector.
crbegin() – Returns a constant reverse iterator pointing to the last element in the
             vector (reverse beginning). It moves from last to                  first
             element
crend() – Returns a constant reverse iterator pointing to the theoretical element
           preceding the first element in the vector (considered as reverse end)*/
```

// C++ program to illustrate the iterators in vector

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

using namespace std;

int main()
{
   vector<int> g1;

   for (int i = 1; i <= 5; i++)
      g1.push_back(i);

   cout << "Output of begin and end: ";
   for (auto it = g1.begin(); it != g1.end(); ++i)
      cout << *it << " ";

   cout << "\nOutput of cbegin and cend: ";
   for (auto i = g1.cbegin(); i != g1.cend(); ++i)
      cout << *i << " ";

   cout << "\nOutput of rbegin and rend: ";
   for (auto ir = g1.rbegin(); ir != g1.rend(); ++ir)
      cout << *ir << " ";

   cout << "\nOutput of crbegin and crend : ";
   for (auto ir = g1.crbegin(); ir != g1.crend(); ++ir)
      cout << *ir << " ";

   return 0;
}
```

Output:

```
Output of begin and end: 1 2 3 4 5
Output of cbegin and cend: 1 2 3 4 5
Output of rbegin and rend: 5 4 3 2 1
Output of crbegin and crend : 5 4 3 2 1
```

Capacity

size() – Returns the number of elements in the vector.
max_size() – Returns the maximum number of elements that the vector can hold.
capacity() – Returns the size of the storage space currently allocated to the vector expressed as number of elements.
resize(n) – Resizes the container so that it contains 'n' elements.
empty() – Returns whether the container is empty.
shrink_to_fit() – Reduces the capacity of the container to fit its size and destroys all elements beyond the capacity.
reserve() – Requests that the vector capacity be at least enough to contain n elements.

// C++ program to illustrate the capacity function in vector

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

using namespace std;

int main()
{
    vector<int> g1;
    for (int i = 1; i <= 5; i++)
        g1.push_back(i);

    cout << "Size : " << g1.size();
    cout << "\nCapacity : " << g1.capacity();
    cout << "\nMax_Size : " << g1.max_size();

    // resizes the vector size to 4
    g1.resize(4);

    // prints the vector size after resize()
    cout << "\nSize : " << g1.size();

    // checks if the vector is empty or not
    if (g1.empty() == false)
        cout << "\nVector is not empty";
    else
        cout << "\nVector is empty";

    // Shrinks the vector
    g1.shrink_to_fit();
    cout << "\nVector elements are: ";
    for (auto it = g1.begin(); it != g1.end(); it++)

        cout << *it << " ";

    return 0;
}
```

Output:

```
Size : 5
Capacity : 8
Max_Size : 4611686018427387903
Size : 4
Vector is not empty
Vector elements are: 1 2 3 4
```

Element access:

reference operator [g] – Returns a reference to the element at position ‘g’ in the vector
at(g) – Returns a reference to the element at position ‘g’ in the vector
front() – Returns a reference to the first element in the vector
back() – Returns a reference to the last element in the vector
data() – Returns a direct pointer to the memory array used internally by the vector to store its owned elements.

// C++ program to illustrate the element accesser in vector

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    vector<int> g1;

    for (int i = 1; i <= 10; i++)
        g1.push_back(i * 10);

    cout << "\nReference operator [g] : g1[2] = " << g1[2];

    cout << "\nat : g1.at(4) = " << g1.at(4);

    cout << "\nfront() : g1.front() = " << g1.front();

    cout << "\nback() : g1.back() = " << g1.back();

    // pointer to the first element
    int* pos = g1.data();

    cout << "\nThe first element is " << *pos;
    return 0;
}
```

<u>Output:</u>

```
Reference operator [g] : g1[2] = 30
at : g1.at(4) = 50
front() : g1.front() = 10
back() : g1.back() = 100
```

```
The first element is 10
```

Modifiers:

- assign() – It assigns new value to the vector elements by replacing old ones
- push_back() – It push the elements into a vector from the back
- pop_back() – It is used to pop or remove elements from a vector from the back.
- insert() – It inserts new elements before the element at the specified position
- erase() – It is used to remove elements from a container from the specified position or range.
- swap() – It is used to swap the contents of one vector with another vector of same type. Sizes may differ.
- clear() – It is used to remove all the elements of the vector container
- emplace() – It extends the container by inserting new element at position
- emplace_back() – It is used to insert a new element into the vector container, the new element is added to the end of the vector

// C++ program to illustrate the Modifiers in vector

```
#include <bits/stdc++.h>
#include <vector>
using namespace std;

int main()
{
    // Assign vector
    vector<int> v;

    // fill the array with 10 five times
    v.assign(5, 10);

    cout << "The vector elements are: ";
    for (int i = 0; i < v.size(); i++)
        cout << v[i] << " ";

    // inserts 15 to the last position
    v.push_back(15);
    int n = v.size();
    cout << "\nThe last element is: " << v[n - 1];

    // removes last element
```

```
v.pop_back();

// prints the vector
cout << "\nThe vector elements are: ";
for (int i = 0; i < v.size(); i++)
    cout << v[i] << " ";

// inserts 5 at the beginning
v.insert(v.begin(), 5);

cout << "\nThe first element is: " << v[0];

// removes the first element
v.erase(v.begin());

cout << "\nThe first element is: " << v[0];

// inserts at the beginning
v.emplace(v.begin(), 5);
cout << "\nThe first element is: " << v[0];

// Inserts 20 at the end
v.emplace_back(20);
n = v.size();
cout << "\nThe last element is: " << v[n - 1];

// erases the vector
v.clear();
cout << "\nVector size after erase(): " << v.size();

// two vector to perform swap
vector<int> v1, v2;
v1.push_back(1);
v1.push_back(2);
v2.push_back(3);
v2.push_back(4);

cout << "\n\nVector 1: ";
for (int i = 0; i < v1.size(); i++)
    cout << v1[i] << " ";
```

```
        cout << "\nVector 2: ";
        for (int i = 0; i < v2.size(); i++)
            cout << v2[i] << " ";

        // Swaps v1 and v2
        v1.swap(v2);

        cout << "\nAfter Swap \nVector 1: ";
        for (int i = 0; i < v1.size(); i++)
            cout << v1[i] << " ";

        cout << "\nVector 2: ";
        for (int i = 0; i < v2.size(); i++)
            cout << v2[i] << " ";
    }
```

Output:

```
The vector elements are: 10 10 10 10 10
The last element is: 15
The vector elements are: 10 10 10 10 10
The first element is: 5
The first element is: 10
The first element is: 5
The last element is: 20
Vector size after erase(): 0

Vector 1: 1 2
Vector 2: 3 4
After Swap
Vector 1: 3 4
Vector 2: 1 2
```

Stack in C++ STL

Stacks are a type of container adaptors with LIFO(Last In First Out) type of working, where a new element is added at one end and (top) an element is removed from that end only.

The functions associated with stack are:
empty() – Returns whether the stack is empty – Time Complexity : O(1)
size() – Returns the size of the stack – Time Complexity : O(1)
top() – Returns the top most element of the stack.
– Time Complexity : O(1)
push(g) – Adds the element ‘g’ at the top of the stack – Time Complexity : O(1)
pop() – Deletes the top most element of the stack – Time Complexity : O(1)

```
// C++ program to demonstrate working of STL stack
          #include <iostream>
          #inluce<stack>
          using namespace std;

          void showstack(stack <int> s)
          {
             while (!s.empty())
             {
                cout << '\t' << s.top();
                s.pop();
             }
             cout << '\n';
          }

          int main ()
          {
             stack <int> s;
             s.push(10);
             s.push(30);
             s.push(20);
             s.push(5);
             s.push(1);

             cout << "The stack is : ";
```

```
        showstack(s);

        cout << "\ns.size() : " << s.size();
        cout << "\ns.top() : " << s.top();


        cout << "\ns.pop() : ";
        s.pop();
        showstack(s);

        return 0;
    }
```

Output:

```
The stack is :    1    5    20    30    10
s.size() : 5
s.top() : 1
s.pop() :    5    20    30    10
```

Queue in Standard Template Library (STL)

Queues are a type of container adaptors which operate in a "first in first out (FIFO) type of arrangement". Elements are inserted at the back (end) and are deleted from the front.

The functions supported by queue are :

empty() – Returns whether the queue is empty.
size() – Returns the size of the queue.
queue::swap() in C++ STL: Exchange the contents of two queues but the queues must be of same type, although sizes may differ.
queue::emplace() in C++ STL: Insert a new element into the queue container, the new element is added to the end of the queue.
queue::front() and queue::back() in C++ STL–
(i)**front()** function returns the first element of the queue.
(ii)**back()** function returns the last element of the queue.
push(g) and pop() –

(i) **push(g)** function adds the element ‘g’ at the end of the queue.
(ii) **pop()** function deletes the first element of the queue.

// CPP code to illustrate Queue in Standard Template Library (STL)

```
#include <iostream>
#include <queue>

using namespace std;

void showq(queue <int> gq)
{
    queue <int> g = gq;
    while (!g.empty())
    {
        cout << '\t' << g.front();
        g.pop();
    }
    cout << '\n';
}

int main()
{
    queue <int> gquiz;
    gquiz.push(10);
    gquiz.push(20);
    gquiz.push(30);

    cout << "The queue gquiz is : ";
    showq(gquiz);

    cout << "\ngquiz.size() : " << gquiz.size();
    cout << "\ngquiz.front() : " << gquiz.front();
    cout << "\ngquiz.back() : " << gquiz.back();

    cout << "\ngquiz.pop() : ";
    gquiz.pop();
    showq(gquiz);

    return 0;
}
```

Output:

```
The queue gquiz is :     10    20    30
gquiz.size() : 3
gquiz.front() : 10
gquiz.back() : 30
gquiz.pop() :     20    30
```

STL Algorithm classes

std::sort() in C++ STL

- C++ STL provides a similar function sort that sorts a vector or array (items with random access).
- Below is a simple program to show working of sort().

// C++ program to demonstrate default behaviour of sort() in STL.

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

int main()
{
    int arr[] = {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};
    int n = sizeof(arr)/sizeof(arr[0]);

    sort(arr, arr+n);

    cout << "\nArray after sorting using default sort is : \n";
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";

    return 0;
}
```

Output :

```
Array after sorting using default sort is :
```

```
0 1 2 3 4 5 6 7 8 9
```

Note: By default, sort() sorts an array in ascending order.

std::reverse() in C++

- reverse() is a predefined function in header file 'algorithm'.
- It is defined as a template in the above mentioned header file.
- It reverses the order of the elements in the range [first, last) of any container.

  Note: The range used is [first,last), which contains all the elements between first and last, including the element pointed by first but not the element pointed by last.

Syntax:

void reverse(BidirectionalIterator first, BidirectionalIterator last)
BidirectionalIterator is an iterator that can be used to access any
elements of a container in both forward and backward direction.

Examples:

Input : 10 11 12 13 14 15 16 17
Output :10 11 12 13 14 17 16 15
Explanation:
reverse(v.begin() + 5, v.begin() + 8);
In the above function, input we have applied reverse() on the vector
from index 5 to index 7.
Therefore when we display the vector we get reverse order
from index 5 to index 7.

// C++ program to illustrate std::reverse() function of STL

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

int main()
{
    vector <int> v ;

    // Inserting elements in vector
    for (int i = 0; i < 8; i++)
        v.push_back(i+10);

    cout << "Reverse only from index 5 to 7 in array:\n";
    // Reversing elements from index 5 to index 7
    reverse(v.begin() + 5, v.begin() + 8);

    // Displaying elements of vector
    vector <int> ::  it;

    for (it = v.begin(); it != v.end(); it++)
        cout << (*it) << " ";

    // Reversing directly from beginning to end
    cout << "\nReverse full array:\n";

    int a[] = {4, 5, 6, 7};
    reverse(begin(a), end(a));

    // Print the array
    cout << a[0] << a[1] << a[2] << a[3] << '\n';
    return 0;
}
```

Output:

```
Reverse only from index 5 to 7 in array:
10 11 12 13 14 17 16 15
Reverse full array:
7 6 5 4
```

min() in C++ STL:

- **std::min** is defined in the header file <algorithm> and 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.

  It can be used in following 3 manners:
  - It compares the two numbers passed in its arguments and **returns the smaller of the two**, and if both are equal, then it returns the first one.
  - It can also compare the two numbers using a **binary function** , which is defined by the user, and then passed as argument in std::min().
  - It is also useful if we want to find the **smallest element in a given list**, and it returns the first one if there are more than one present in the list.

The three versions are as defined below:

1. For comparing elements using **<** :

Syntax:

T& min (T& a, T& b);
a and b are the numbers to be compared.
Returns: Smaller of the two values.

```
// C++ program to demonstrate the use of std::min()
            #include <iostream>
            #include <algorithm>
            using namespace std;
            int main()
            {
               int a = 5;
               int b = 7;
               cout << min(a, b) << "\n";

               // Returns the first one if both the numbers are same
               cout << min(7, 7);

               return 0;
            }
```

Output:

```
5
7
```

2.For finding the minimum element in a list:

Syntax:

template T min (initializer_list li);

li: An initializer_list object.
Returns: Smallest of all the values.

// C++ program to demonstrate the use of std::min

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

int main()
{

   // Finding the smallest of all the numbers
   cout << min({ 1, 2, 3, 4, 5, 0, -1, 7 }) << "\n";

   return 0;
}
```

Output:

```
-1
```

max() in C++ STL:

- **max** is defined in the header file <algorithm> and is used to find out the largest of the number passed to it.
- It returns the first maximum value of them, if there are more than one.

1.For comparing elements using a pre-defined function:
Syntax:

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

Here, a and b are the numbers to be compared.

Returns: Larger of the two values.

// C++ program to demonstrate the use of std::max

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

// Defining the binary function
int main()
{
    int a = 7;
    int b = 28;

    cout << max(a,b) << "\n";

    // Returns the first one if both the numbers are same
    cout << max(7,7);

return 0;
}
```

Output:

```
28
7
```

2.For finding the maximum element in a list:

Syntax:

template T max (initializer_list li);

Here, comp is optional and can be skipped.
li: An initializer_list object.
Returns: Largest of all the values.

// C++ program to demonstrate the use of std::max

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

int main()
{

    // Finding the largest of all the numbers
    cout << max({1, 2, 3, 4, 5, 10, -1, 7}) << "\n";
return 0;
}
```

<u>Output:</u>

```
10
```