About the content

This C++ material taken from the following websites:

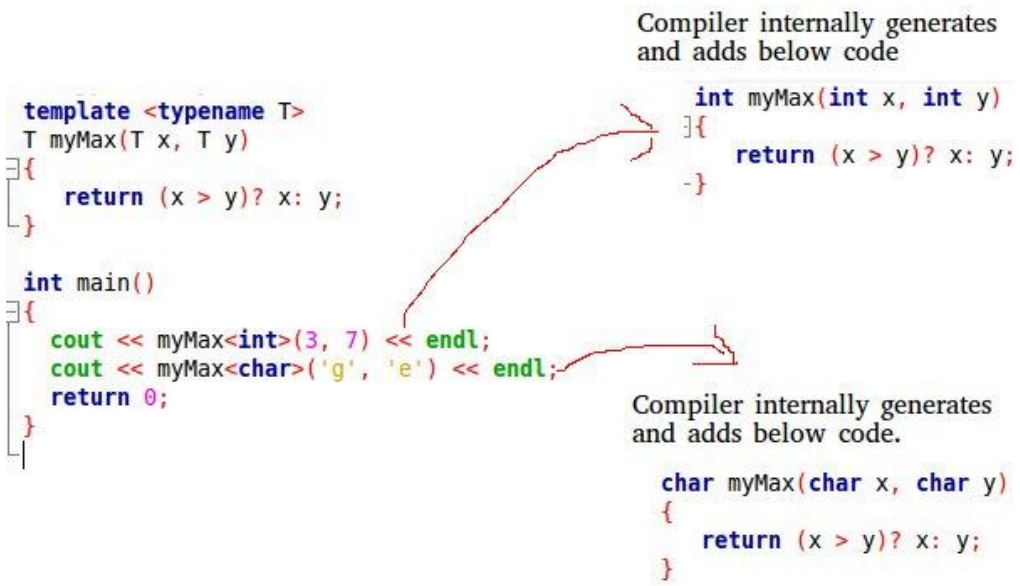
- https://www.tutorialspoint.com/cplusplus
- https://www.geeksforgeeks.org/cpp-tutorial
- https://www.javatpoint.com/cpp-tutorial
- https://www.programiz.com/cpp-programming
- https://www.cplusplus.com

UNIT-V
Module-1

Function and class Templates in C++

A **template** is a simple yet very powerful tool in C++. 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. For example, a software company may need to sort() for different data types. Rather than writing and maintaining multiple codes, we can write one sort() and pass data type as a parameter.
C++ adds two new keywords to support templates: *'template'* and *'typename'*. The second keyword can always be replaced by the keyword 'class'.

How Do Templates Work?
Templates are expanded at compiler time. This is like macros. The difference is, that the compiler does type checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of the same function/class.



What is Generics in C++

Generics is the idea to allow type (Integer, String, … etc and user-defined types) to be a parameter to functions, classes and interfaces. For example, classes like an array, map, etc, which can be used using generics very efficiently. We can use them for any type.

The method of Generic Programming implemented to increase the efficiency of the code. Generic Programming enables the programmer to write a general algorithm which will work with all data types. It eliminates the need to create different algorithms if the data type is an integer, string or a character.

The advantages of Generic Programming are

1. Code Reusability
2. Avoid Function Overloading
3. Once written it can be used for multiple times and cases.

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

// One function works for all data types.  This would work
// even for user defined types if operator '>' is overloaded
template <typename T> T myMax(T x, T y)
{
   return (x > y) ? x : y;
}

int main()
{
   cout << myMax<int>(3, 7) << endl; // Call myMax for int
   cout << myMax<double>(3.0, 7.0)
        << endl; // call myMax for double
   cout << myMax<char>('g', 'e')
        << endl; // call myMax for char

   return 0;
}
```

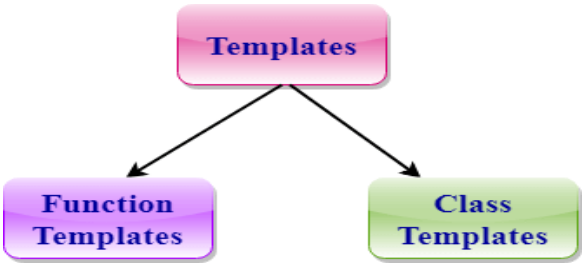
Output

```
7
7
G
```

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 Templates:
We write a generic function that can be used for different data types For example, if we have an add() function, we can create versions of the add function for adding the int, float or double type values.

Class Template:
We can define a template for a class. For example, a class template can be created for the array class that can accept the array of various types such as int array, float array or double array. Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc

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

```
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.
Let's see a simple example of a function template:

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

}
int main()
{
 int i =2,  j =3,k;
 float m = 2.3,  n = 1.2, r;
 k= add(i,j);
```

```
    r= add(m,n);

    cout<<"Addition of i and j is :"<< k <<endl;
    cout<<"Addition of m and n is :"<< r <<endl;
    return 0;
}
```

Output:

```
Addition of i and j is :5
Addition of m and n is :3.5
```

In the above example, we create the function template which can perform the addition operation on any type either it can be integer, float or double.

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.

Let's see a simple example:

```
#include <iostream>
using namespace std;
template<class X,class Y> void fun(X a,Y b)
{
   cout << "Value of a is : " <<a<< endl;
   cout << "Value of b is : " <<b<< endl;
}

int main()
{
  fun(15,12.3);

  return 0;
}
```

Output:

```
Value of a is : 15
Value of b is : 12.3
```

In the above example, we use two generic types in the template function, i.e., X and Y.

Below is the program to implement Bubble Sort using templates in C++:

```
// CPP code for bubble sort using template function
#include <iostream>
using namespace std;

/*A template function to implement bubble sort. We can use this for any data type that
supports comparison operator < and swap works for it.*/

template <class T> void bubbleSort(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]);
}

// Driver Code
int main()
{
    int a[5] = { 10, 50, 30, 40, 20 };
    int n = sizeof(a) / sizeof(a[0]);

    // calls template function
    bubbleSort<int>(a, n);

    cout << " Sorted array : ";
    for (int i = 0; i < n; i++)
        cout << a[i] << " ";
    cout << endl;

    return 0;
}
```

Output

Sorted array : 10 20 30 40 50

Overloading a Function Template

We can overload the generic function means that the overloaded template functions can differ in the parameter list.

Let's understand this through a simple example:

```
#include <iostream>
using namespace std;
template<class X> void fun(X  a)
{
    cout << "Value of a is : " <<a<< endl;
}
template<class X,class Y> void fun(X b ,Y c)
{
    cout << "Value of b is : " <<b<< endl;
    cout << "Value of c is : " <<c<< endl;
}
int main()
{
   fun(10);
   fun(20,30.5);
   return 0;
}
```

Output:

```
Value of a is : 10
Value of b is : 20
Value of c is : 30.5
```

In the above example, template of fun() function is overloaded.

Restrictions of Generic Functions

Generic functions perform the same operation for all the versions of a function except the data type differs. Let's see a simple example of an overloaded function which cannot be replaced by the generic function as both the functions have different functionalities.

Let's understand this through a simple example:

```
#include <iostream>
using namespace std;
void fun(double a)
{
   cout<<"value of a is : "<<a<<'\n';
}

void fun(int b)
{
   if(b%2==0)
   {
      cout<<"Number is even";
   }
   else
   {
      cout<<"Number is odd";
   }
}

int main()
{
  fun(4.6);
  fun(6);
  return 0;
}
```

Output:

```
value of a is : 4.6
Number is even
```

In the above example, we overload the ordinary functions. We cannot overload the generic functions as both the functions have different functionalities. First one is displaying the value and the second one determines whether the number is even or not.

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;
```

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.

Let's see a simple example:

```
#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;
}
```

Output:

```
Addition of num1 and num2 : 11
```

In the above example, we create a template for class A. Inside the main() method, we create the instance of class A named as, 'd'.

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.
}
```

Let's see a simple example when class template contains two generic data types.

```
#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;
}
```

Output:

```
Values of a and b are : 5,6.5
```

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.

Let' s see the following 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.
```

Let's see a simple example of nontype template arguments.

```
#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++)
    {
      cout << arr[i] << " ";
    }
  }
};
int main()
{
  A<int,10> t1;
  t1.insert();
```

```
    t1.display();
    return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

In the above example, the class template is created which contains the nontype template argument, i.e., size. It is specified when the object of class 'A' is created.

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.

MODULE-2

The C++ Standard Template Library (STL)

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized. Working knowledge of template classes is a prerequisite for working with STL.

STL has 4 components:

- **Algorithms**
- **Containers**
- **Iterators**

Algorithms

The header algorithm defines a collection of functions specially designed to be used on a range of elements. They act on containers and provide means for various operations for the contents of the containers.

- Algorithm
  - Sorting
  - Searching
  - Important STL Algorithms
  - Useful Array algorithms
  - Partition Operations
- Numeric
  - valarray class

Containers

Containers or container classes store objects and data. There are in total seven standards "first-class" container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

- Sequence Containers: implement data structures that can be accessed in a sequential manner.
    - vector
    - list
    - deque
    - arrays
    - forward_list(Introduced in C++11)
- Container Adaptors: provide a different interface for sequential containers.
    - queue
    - priority_queue
    - stack
- Associative Containers: implement sorted data structures that can be quickly searched (O(log n) complexity).
    - set
    - multiset
    - map
    - multimap
- Unordered Associative Containers: implement unordered data structures that can be quickly searched
    - unordered_set (Introduced in C++11)
    - unordered_multiset (Introduced in C++11)
    - unordered_map (Introduced in C++11)
    - unordered_multimap (Introduced in C++11)

Iterators

As the name suggests, iterators are used for working upon a sequence of values. They are the major feature that allows generality in STL.

- Iterators

Algorithms

Sort in C++ (STL)

Sorting is one of the most basic functions applied to data. It means arranging the data in a particular fashion, which can be increasing or decreasing. There is a builtin function in C++ STL by the name of sort().
This function internally uses IntroSort. In more details it is implemented using hybrid of QuickSort, HeapSort and InsertionSort.By default, it uses QuickSort but if QuickSort is doing unfair partitioning and taking more than N*logN time, it switches to HeapSort and when the array size becomes really small, it switches to InsertionSort.

The prototype for sort is :
sort(startaddress, endaddress)

startaddress: the address of the first element of the array
endaddress: the address of the next contiguous location of the last element of the array.
So actually sort() sorts in the range of [startaddress,endaddress)

```
// C++ program to sort an array
#include <algorithm>
#include <iostream>

using namespace std;

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

// Driver code
int main()
{
   int a[] = { 1, 5, 8, 9, 6, 7, 3, 4, 2, 0 };

   // size of the array
   int asize = sizeof(a) / sizeof(a[0]);
   cout << "The array before sorting is : \n";

   // print the array
   show(a, asize);

    // sort the array
   sort(a, a + asize);

   cout << "\n\nThe array after sorting is :\n";

   // print the array after sorting
   show(a, asize);

   return 0;
}
```

Output

```
The array before sorting is :
1 5 8 9 6 7 3 4 2 0

The array after sorting is :
0 1 2 3 4 5 6 7 8 9
```

How to sort in descending order?

sort() takes a third parameter that is used to specify the order in which elements are to be sorted. We can pass the “greater()” function to sort in descending order. This function does a comparison in a way that puts greater elements before.

```
// C++ program to demonstrate descending order sort using greater<>().
```

```
#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, greater<int>());

   cout << "Array after sorting : \n";
   for (int i = 0; i < n; ++i)
      cout << arr[i] << " ";

   return 0;
}
```

Output

Array after sorting :
9 8 7 6 5 4 3 2 1 0

Binary Search in C++ Standard Template Library (STL)

Binary search is a widely used searching algorithm that requires the array to be sorted before search is applied. The main idea behind this algorithm is to keep dividing the array in half (divide and conquer) until the element is found, or all the elements are exhausted.
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. If the middle term is less than the target, the search is performed in the right sub-array.

The prototype for binary search is :

binary_search(startaddress, endaddress, valuetofind)

Parameters :
startaddress: the address of the first element of the array.
endaddress: the address of the next contiguous location of the last element of the array.
valuetofind: the target value which we have to search for.
Returns :
true if an element equal to valuetofind is found, else false.

```
// CPP program to implement Binary Search in Standard Template Library (STL)
#include <algorithm>
#include <iostream>

using namespace std;

void show(int a[], int arraysize)
{
```

```
    for (int i = 0; i < arraysize; ++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);

    cout << "\n\nLet's say we want to search for ";
    cout << "\n2 in the array So, we first sort the array";
    sort(a, a + asize);
    cout << "\n\nThe array after sorting is : \n";
    show(a, asize);
    cout << "\n\nNow, we do the binary search";
    if (binary_search(a, a + 10, 2))
        cout << "\nElement found in the array";
    else
        cout << "\nElement not found in the array";

    cout << "\n\nNow, say we want to search for 10";
    if (binary_search(a, a + 10, 10))
        cout << "\nElement found in the array";
    else
        cout << "\nElement not found in the array";

    return 0;
}
```

Output

```
The array is :
1,5,8,9,6,7,3,4,2,0,

Let's say we want to search for
2 in the array So, we first sort the array

The array after sorting is :
0,1,2,3,4,5,6,7,8,9,

Now, we do the binary search
Element found in the array

Now, say we want to search for 10
Element not found in the array
```

Containers

Vector in C++ (STL)

Vectors are the 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 the array may need to be extended. 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.

Declaration of Vectors in C++

It is mandatory to include #include<vector> library before using vectors in C++.
For Vector declaration we need to follow the below syntax:

vector< object_type > vector_variable_name;

Initialization of Vectors

1. Pushing the values one-by-one in vector using push_back():

- All the elements that need to be stored in the vector are pushed back one-by-one in the vector using the push_back() method.

Syntax:

vector_name.push_back(element_value);

2. Using Array:

- This method uses array as a parameter to be passed in the vector constructor.

Syntax:

vector<object_type> vector_name {val1,val2,val3,....,valn};

3.Using already initialized vector:

- This method uses an already created vector to create a new vector with the same values.
- This method passes the begin() and end() of an already initialized vector.

Syntax:

vector<object_type> vector_name_1{val1,val2,…,valn};
vector<object_type> vector_name_2(vector_name_1.begin(),vector_name_1.end())

Various Functions in Vectors are

Iterators:

1. begin() – Returns an iterator pointing to the first element in the vector
2. end() – Returns an iterator pointing to the theoretical element that follows the last element in the vector
3. rbegin() – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
4. rend() – Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)
5. cbegin() – Returns a constant iterator pointing to the first element in the vector.
6. cend() – Returns a constant iterator pointing to the theoretical element that follows the last element in the vector.
7. crbegin() – Returns a constant reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element

8. 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: ";
//u can use auto i instead of vector<int>::iterator  i
   for (vector<int>::iterator  i = g1.begin(); i != g1.end(); i++)
      cout << *i << " ";

   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

1. size() – Returns the number of elements in the vector.
2. max_size() – Returns the maximum number of elements that the vector can hold.
3. capacity() – Returns the size of the storage space currently allocated to the vector expressed as number of elements.
4. resize(n) – Resizes the container so that it contains ‘n’ elements.
5. empty() – Returns whether the container is empty.
6. shrink_to_fit() – Reduces the capacity of the container to fit its size and destroys all elements beyond the capacity.
7. 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:

1. reference operator [g] – Returns a reference to the element at position ‘g’ in the vector
2. at(g) – Returns a reference to the element at position ‘g’ in the vector
3. front() – Returns a reference to the first element in the vector
4. back() – Returns a reference to the last element in the vector

5. 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 access in vector
#include <iostream >
#include <vector>

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;
}
```

Output:

```
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:

1. assign() – It assigns new value to the vector elements by replacing old ones
2. push_back() – It push the elements into a vector from the back
3. pop_back() – It is used to pop or remove elements from a vector from the back.
4. insert() – It inserts new elements before the element at the specified position
5. erase() – It is used to remove elements from a container from the specified position or range.
6. swap() – It is used to swap the contents of one vector with another vector of same type. Sizes may differ.
7. clear() – It is used to remove all the elements of the vector container
8. emplace() – It extends the container by inserting new element at position
9. 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 <iostream>
#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
```

When to use Vectors?

- We can use Vectors in the following circumstances:
- It is advisable to use vectors when data are consistently changing.
- If the size of data is unknown then it is advisable to use vectors.
- It is advisable to use vectors when elements are not predefined.
- Compared to arrays there are more ways to copy vectors..

Vectors are sequence containers having the ability to resize themselves. In this tutorial on C++ vectors, you have learned the different member functions of vectors, their functionalities, and the difference between vectors and arrays.

List in C++ (STL)

Lists are sequence containers that allow non-contiguous memory allocation. As compared to vector, the list has slow traversal, but once a position has been found, insertion and deletion are quick.

Normally, when we say a List, we talk about a doubly linked list. For implementing a singly linked list, we use a forward list. Below is the program to show the working of some functions of List:

```
// CPP program to show the implementation of List
#include <iostream>
#include <iterator>
#include <list>
using namespace std;

// function for printing the elements in a list
void showlist(list<int> g)
{
   list<int>::iterator it;
   for (it = g.begin(); it != g.end(); ++it)
      cout << '\t' << *it;
   cout << '\n';
}

// Driver Code
int main()
{

   list<int> gqlist1, gqlist2;

   for (int i = 0; i < 10; ++i) {
      gqlist1.push_back(i * 2);
      gqlist2.push_front(i * 3);
   }
   cout << "\nList 1 (gqlist1) is : ";
   showlist(gqlist1);

   cout << "\nList 2 (gqlist2) is : ";
   showlist(gqlist2);

   cout << "\ngqlist1.front() : " << gqlist1.front();
   cout << "\ngqlist1.back() : " << gqlist1.back();

   cout << "\ngqlist1.pop_front() : ";
   gqlist1.pop_front();
   showlist(gqlist1);

   cout << "\ngqlist2.pop_back() : ";
   gqlist2.pop_back();
   showlist(gqlist2);

   cout << "\ngqlist1.reverse() : ";
   gqlist1.reverse();
   showlist(gqlist1);

   cout << "\ngqlist2.sort(): ";
   gqlist2.sort();
   showlist(gqlist2);

   return 0;
}
```

Output

```
List 1 (gqlist1) is :    0   2   4   6   8   10   12   14   16   18

List 2 (gqlist2) is :    27   24   21   18   15   12   9   6   3   0

gqlist1.front() : 0
gqlist1.back() : 18
gqlist1.pop_front() :    2   4   6   8   10   12   14   16   18

gqlist2.pop_back() :    27   24   21   18   15   12   9   6   3

gqlist1.reverse() :    18   16   14   12   10   8   6   4   2

gqlist2.sort():    3   6   9   12   15   18   21   24   27
```

Functions Used with List

Method/Function	Description
insert()	It inserts the new element before the position pointed by the iterator.
push_back()	It adds a new element at the end of the list.
push_front()	It adds a new element to the front.
pop_back()	It deletes the last element.
pop_front()	It deletes the first element.
empty()	It checks whether the list is empty or not.
size()	It finds the number of elements present in the list.
max_size()	It finds the maximum size of the list.
front()	It returns the first element of the list.
back()	It returns the last element of the list.
swap()	It swaps two list when the type of both the list are same.
reverse()	It reverses the elements of the list.
sort()	It sorts the elements of the list in an increasing order.
merge()	It merges the two sorted list.
splice()	It inserts a new list into the invoking list.
unique()	It removes all the duplicate elements from the list.
resize()	It changes the size of the list container.
assign()	It assigns a new element to the list container.
emplace()	It inserts a new element at a specified position.
emplace_back()	It inserts a new element at the end of the vector.

emplace_front()	It inserts a new element at the beginning of the list.

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 (top) and an element is removed from that end only. Stack uses an encapsulated object of either vector or deque (by default) or list (sequential container class) as its underlying container, providing a specific set of member functions to access its elements.

Stack Syntax:-

For creating a stack, we must include the <stack> header file in our code. We then use this syntax to define the std::stack:

template <class Type, class Container = deque<Type> > class stack;

Template Parameters

Type – is the Type of element contained in the std::stack. It can be any valid C++ type or even a user-defined type.

Container – is the Type of underlying container object.

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 a reference to 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)

```
#include <iostream>
#include <stack>
using namespace std;
int main() {
   stack<int> stack;
   stack.push(21);
   stack.push(22);
   stack.push(24);
   stack.push(25);

      stack.pop();
   stack.pop();

   while (!stack.empty()) {
      cout << ' ' << stack.top();
      stack.pop();
   }
}
```

Output

22 21

Code Explanation:

1. Include the iostream header file or <bits/stdc++.h> in our code to use its functions.
2. Include the stack header file in our code to use its functions if already included <bits/stdc++.h> then no need of stack header file because it has already inbuilt function in it.
3. Include the std namespace in our code to use its classes without calling it.
4. Call the main() function. The program logic should be added within this function.
5. Create a stack to store integer values.
6. Use the push() function to insert the value 21 into the stack.
7. Use the push() function to insert the value 22 into the stack.
8. Use the push() function to insert the value 24 into the stack.
9. Use the push() function to insert the value 25 into the stack.
10. Use the pop() function to remove the top element from the stack, that is, 25. The top element now becomes 24.
11. Use the pop() function to remove the top element from the stack, that is, 24. The top element now becomes 22.
12. Use a while loop and empty() function to check whether the stack is NOT empty. The ! is the NOT operator.
13. Printing the current contents of the stack on the console.
14. Call the pop() function on the stack.
15. End of the body of the while loop.
16. End of the main() function body.

Queue in C++

In computer science we go for working on a large variety of programs. Each of them has their own domain and utility. Based on the purpose and environment of the program creation, we have a large number of data structures available to choose from. One of them is 'queues. Before discussing about this data type let us take a look at its syntax.

Syntax

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

This data structure works on the FIFO technique, where FIFO stands for First In First Out. The element which was first inserted will be extracted at the first and so on. There is an element called as 'front' which is the element at the front most position or say the first position, also there is an element called as 'rear' which is the element at the last position. In normal queues insertion of elements take at the rear end and the deletion is done from the front.

Queues in the application areas are implied as the container adaptors.

The containers should have a support for the following list of operations:

- empty
- size
- push_back
- pop_front
- front
- back

Template Parameters

T: The argument specifies the type of the element which the container adaptor will be holding.
Container: The argument specifies an internal object of container where the elements of the queues are held.

The functions associated with Queue are:

With the help of functions, an object or variable can be played with in the field of programming. Queues provide a large number of functions that can be used or embedded in the programs. A list of the same is given below:

Function	Description
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.

Example: A simple program to show the use of basic queue functions.

```
#include <iostream>
#include <queue>
using namespace std;
void showsg(queue <int> sg)
{
   queue <int> ss = sg;
   while (!ss.empty())
   {
      cout << '\t' << ss.front();
      ss.pop();
   }
   cout << '\n';
}

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

   cout << "The queue fquiz is : ";
   showsg(fquiz);

   cout << "\nfquiz.size() : " << fquiz.size();
   cout << "\nfquiz.front() : " << fquiz.front();
   cout << "\nfquiz.back() : " << fquiz.back();

   cout << "\nfquiz.pop() : ";
```

```
    fquiz.pop();
    showsg(fquiz);

    return 0;
}
```

Output:

```
The queue fquiz is :    10        20        30

fquiz.size() : 3
fquiz.front() : 10
fquiz.back() : 30
fquiz.pop() :           20        30
```

This C++ material taken from the following websites:

- https://www.tutorialspoint.com/cplusplus
- https://www.w3schools.com/CPP
- https://www.geeksforgeeks.org/cpp-tutorial
- https://www.javatpoint.com/cpp-tutorial
- https://www.programiz.com/cpp-programming
- https://www.cplusplus.com
- https://www.mygreatlearning.com/