# Templates and STL

## Today’s Agenda

- Introduction to templates
- Function template
- class template
- Inheritance in template class(single level)
- Introduction to STL
- Containers
- Algorithms and iterators
- Container - Vector and List.

## Let’s Get Started-

## Introdution

Note: Encourage students to answers the following questions.

1. What are templates?

Ans: In general these are standard forms .

1. Have you all used a templates in your life?

Ans: Yes. We all have used templates somewhere in our life

1. Give example.

Ans: Suppose you go to bank to deposit or withdraw money. In earlier times, we had to fill the withdrawal or deposit form where we put necessary information like name, account number, amount, etc. Imagine what will happen if the form is not available? Everyone will try to provide the information in his / her own format. This form is called template.

Other examples of templates could be exam form, admission form, etc.

## Why templates

Let us understand this with the help of example.

```
#include<iostream>
using namespace std;
int Max( int a, int b){
		return a>b ? a :b;
}
int main(){
		cout<< "Max of two numbers : " <<Max(4,5);
		return 0;
}
```

The above program gives max to two integers. What if we want to use it for doubles?

## Why templates

For doubles, we have to overload the function.

```
#include<iostream>
using namespace std;
int Max( int a, int b){
		return a>b ? a :b;
}
double Max( double a, double b){
		return a>b ? a :b;
}
int main(){
		cout<<Max(4,5)<<endl;
		cout<<Max(4.5, 3.2)<<endl;
		return 0;
}
```

## Why Templates

Similarly if we want to find out max of two strings , we have to modify the function to handle strings.

So for any new data type we want to use the same function, we have to overload the function with this new data type.

Instead we can make life simple by writing code in a way that is independent of any particular type.

This concept is called as template. A template is a simple and 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.

## Templates

A template is a blueprint or formula for creating a generic class or a function

C++ supports two types of templates: **1. Function 2. Class**

Template is essential feature added recently to C++.

This new concept allows programmers to define generic classes and functions and thus provide support for generic programming.

Generic programming is an approach of C++ programming where generic types are used as parameters so that they can work for various cases of suitable data type and data structure.

The template is the basis for establishing the concept of generic programming, which entails writing code in a way that is independent of a particular type.

## Function template

## Practice question

```
#include <iostream>
using namespace std;
template <class X>  //additional line for template syntax
X Max (X a, X b) {  //X is any data type
//The above two lines could be written on same or different lines as shown below
//template <class X> X Max (X a, X b){
return a >b ? a:b;
}
int main(){
		cout<<Max(4,5)<<endl;
		cout<<Max(4.5, 3.2)<<endl;
		return 0;
}
```

Output:

```
5
4.5
```

## Practice question

Explanation:
In addition to normal program, we have preceded the function definition with template <class T> which we always do.

X Max (X a, X b):  is a function prototype where X could be any name you like. X works as placeholder written in <>. If you pass int as data type, X will be int. If you pass double as data type, X will be double.

## syntax

A function template is also known as generic function.

Syntax:

```
template <class type> ret-type function_name(parameter list)
{
// body of function
}
```

template: is a keyword and is to be written in small case

class is a keyword which is madatory

Type is a placeholder which will be replaced with data type you pass from calling function.

Ret-type function_name (parameter list) : is a normal function prototype.

## Practice question

What if I have different types of arguments in function call?

The above program can be modified with reference to the following syntax:

template <class type1, class type2> type1 or type2 function_name (type1 arg1, type2 arg2, ....)

Let us understand it better with the help of example.

**Hint: Initially, Always write the program in usual way and then modify it to make it generic.**

## Practice question

```
#include <iostream>
using namespace std;
template <class T, class M> M Max(T a, M b)
{
		return a>b ? a :b;
}
int main()
{
		cout<<(3,4.6)<<endl;
		cout<<(4.5, 4)<<endl;
		return 0;
}
```

Output:

```
4.6
4
```

//second output is 4 as first argument is truncated from double to int as 4 .

## Practice question

Write a function template to accept an array and its size and return sum of elements of an array.
Let us first write the simple program using function. Then convert to generic program.

```
#include <iostream>
using namespace std;
int Sum (int a[], int size)
{
		int s=0;
		for (int i=0; i<size;i++)
		{
				s=s+a[i];
		}
		return s;
}
```

## Practice question

```
int main()
{
        int x[]={10,20,30,40,50};
        //double y[]={ 1.1,2.2,3.3};
        cout<<"Sum of int array="<<Sum(x ,5);
}
```

Output:

```
Sum of int array=150
```

## Practice question

Let us now modify the above program using template

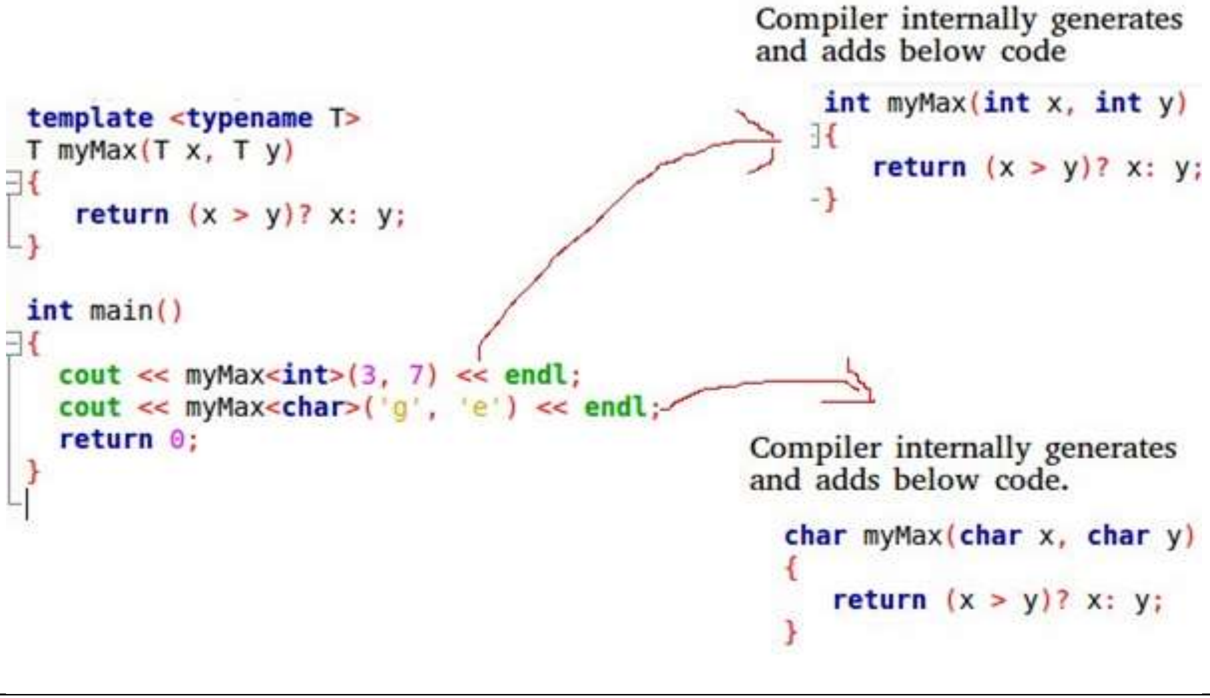
```
template <class T>T Sum (T a[], int size)
{
        T s=0;
        for (int i=0; i<size;i++)
        {
                s=s+a[i];
        }
        return s;
}
int main() {
        int x[]={10,20,30,40,50};
        double y[]={ 1.1,2.2,3.3};
        cout<<"Sum of int array="<<Sum(x ,5)<<endl;
        cout<<"Sum of double array= "<<Sum(y,3)<<endl;
}
```

Output:

```
Sum of int array=150
Sum of double array=6.6
```

## How templates work

Templates are expanded at compiler time. This is like macros.
The difference is, 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 same function/class.



```
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}

int main()
{
   cout << myMax<int>(3, 7) << endl;
   cout << myMax<char>('g', 'e') << endl;
   return 0;
}
```

Compiler internally generates and adds below code

```
int myMax(int x, int y)
{
    return (x > y)? x: y;
}
```

Compiler internally generates and adds below code.

```
char myMax(char x, char y)
{
    return (x > y)? x: y;
}
```

## Assignment

Create a template function for addition of two numbers. This function returns the sum of the numbers to calling function. Pass different types of data to it :
(int, int)
(double, double)
(int , double)
(double, int)

Observe the advantages of template with respect to function overloading.

## Class template

## Class template

Class Templates: Like function templates, class templates are useful when a class defines something that is independent of the data type.

Syntax:

```
template <class type>
class class-name
{
            …
            …
            …
 }
```

Here, type is the placeholder type name, which will be specified when a class is instantiated.

## Practice question

Let is create a class with 2 data members and a constructor

```
class A  {
    int x;
    int y;
public:
    A() {    cout<<"Constructor Called"<<endl;    }
};

int main()  {
   A a;
   return 0;
}
```

Output: Constructor Called

## Practice question

**Let us modify it to display the sum.**

```
class A
{
        int x;
        int y;
        public:   A()
        {
                x=10;      y=2;
                cout<<"Constructor Called"<<endl;
        }
        int display()
        {
                return x+y;
        }
};
```

```
int main()
 {
        A a;
        cout<<a.display();
        return 0;
}
```

**Output:**

```
Constructor Called
12
```

## Practice question

**Let us modify to make it generic class**

```
Template <class T>
class A
{
        T x;
        T y;
        public:   A()
        {
                x=10;      y=2;
                cout<<"Constructor Called"<<endl;
        }
        T display()
        {
                return x+y;
        }
};
```

```
int main()
 {
        A <int> a;
        cout<<a.display();
        return 0;
}
```

**Output:**

```
Constructor Called
12
```

## Practice question

Explanation:

- To make generic class, we precede class definition with Template <class T> .
- In class definition, whichever data type we want to make generic , we will replace it with T. Here int is replaced with T.
- T x;    T y; // replaced int x;int y;
- Also int display() replaced with T display() as we want the function to return the int value of sum of the variables.
- In main(), usually we create objects using A a; ( i.e. classname objectname). In this case, memory is allocated to objects depending on types of data members.
- But since we declared variables of type T, compiler does not know how much memory to allocate for object ‘a’ as T is unknown. Hence to give hint to compiler, we must create objects using following syntax:
- A <int> a; //will create objects and allocate memory for integers
- A <double > b; //will create objects and allocate memory for doubles.

## Practice question

**Let us modify it to have parameterized constructor**

```
Template <class T>
class A
{
        T x;
        T y;
        public:   A(T m, T n)
        {
                x=m;      y=n;
                cout<<"Constructor Called"<<endl;
        }
        T display()
        {
                return x+y;
        }
};
```

```
int main()
 {
        A <int> a(2,3);
        cout<<a.display()<<endl;
        A <double> a(3.4, 5.7);
        cout<<a.display()<<endl;


        return 0;
}
```

**Output:**

```
Constructor Called
5
Constructor Called
9.1
```

## Practice question

**Let us modify it to handle two different data types**

```
Template <class T, class W>
class A
{
        T x;
        W y;
        public:   A(T m, W n)          {
                   x=m;     y=n;
                   cout<<"Constructor Called"<<endl;
        }
        double display()   //always want double
irrespective of input
        {
                   return x+y;
        }
};
```

```
int main()
 {
            A <int, double> a(2,3.4);
            cout<<a.display()<<endl;
            A <double,int> a(3.4, 5);
            cout<<a.display()<<endl;
            return 0;
}
```

**Output:**

```
Constructor Called
5.4
Constructor Called
8.4
```

## Practice question

The only member function in the previous class template has been defined inline within the class declaration itself.
In case that we define a function member outside the declaration of the class template, we must always precede that definition with the template <...> prefix:
Let us see how the member function definition will be modified.

## Practice question

**Let us modify it to have the member function definition outside the class**

```
Template <class T>
class Myclass
{
        T x;
        T y;
        public:   Myclass(T m, T n)
        {
                x=m;     y=n;
                cout<<"Constructor Called"<<endl;
        }
        T display() ;
};
```

```
template <class T>
T Myclass <T> :: display()
{
        return x+y;
}

int main()
 {
        Myclass <int> a(2,3);
        cout<<a.display()<<endl;
        return 0;
}
```

**Output:**

```
Constructor Called
5
```

## Practice question

**Explanation**:

```
T Myclass :: display() //ideally this would have been the definition for generic member .
{
		return x+y;
}
```

**template <class T>   //but highlighted below are the additional changes**

T Myclass **<T>** :: display()

```
{
		return x+y;
}
```

Confused by so many T's? There are three T's in this declaration: The first one is the template parameter. The second T refers to the type returned by the function. And the third T (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.

## Assignment

Provide the definition for the function display() out the class.

```
Template <class T, class W>
class A
{
        T x;
        W y;
        public:   A(T m, W n)         {
                  x=m;     y=n;
                  cout<<"Constructor Called"<<endl;
        }
        double display() ;
};
// display()???
```

```
int main()
 {
        A <int, double> a(2,3.4);
        cout<<a.display()<<endl;
        A <double,int> a(3.4, 5);
        cout<<a.display()<<endl;
        return 0;
}
```

**Output:**

```
Constructor Called
5.4
Constructor Called
8.4
```

## Practice question

Which of the following is true about templates.
1) Template is a feature of C++ that allows us to write one code for different data types.

2) We can write one function that can be used for all data types including user defined types.

3) We can write one class or struct that can be used for all data types including user defined types.

4) Template is an example of compile time polymorphism.

Options:
1. 1,2
2. 1,2,3
3. 1,2,4
4. 1,2,3,4

## Practice question

Which of the following is true about templates.
1) Template is a feature of C++ that allows us to write one code for different data types.

2) We can write one function that can be used for all data types including user defined types.

3) We can write one class or struct that can be used for all data types including user defined types.

4) Template is an example of compile time polymorphism.

Options:
1. 1,2
2. 1,2,3
3. 1,2,4
4. 1,2,3,4

## Practice question

What will be the output of the following program?

```
template <typename T>
void fun(const T&x)
{
    static int count = 0;
    cout << "x = " << x << " count = " << count << endl;
    ++count;
    return;
}
```

```
int main()
{
    fun<int> (1);
    cout << endl;
    fun<int>(1);
    cout << endl;
    fun<double>(1.1);
    cout << endl;
    return 0;
}
```

## Practice question

```
x = 1 count = 0

x = 1 count = 1

x = 1.1 count = 0
```

Compiler creates a new instance of a template function for every data type. So compiler creates two functions in the above example, one for int and other for double. Every instance has its own copy of static variable. The int instance of function is called twice, so count is incremented for the second call.

## Practice question

Which of the following is correct about templates in C++?
(1)When we write overloaded function we must code the function for each usage.
(2)When we write function template we code the function only once.

1. 1 only
2. 2 only
3. Both are true
4. Both are false

## Practice question

Which of the following is correct about templates in C++?
(1)When we write overloaded function we must code the function for each usage.
(2)When we write function template we code the function only once.

1. 1 only
2. 2 only
3. Both are true
4. Both are false

## Practice question

What will be the output of the following ?

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

template <typename T>
T max(T x, T y)
{
    return (x > y)? x : y;
}
int main()
{
    cout << max(3, 7) << std::endl;
    cout << max(3.0, 7.0) << std::endl;
    cout << max(3, 7.0) << std::endl;
    return 0;
}
```

**Output:**

1. Compiler error in all cout statements
2. Compiler error in last cout statements
3. 7
   7.0
   7.0
4. Runtime error

## Practice question

What will be the output of the following ?

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

template <typename T>
T max(T x, T y)
{
    return (x > y)? x : y;
}
int main()
{
    cout << max(3, 7) << std::endl;
    cout << max(3.0, 7.0) << std::endl;
    cout << max(3, 7.0) << std::endl;
    return 0;
}
```

**Output:**

1. Compiler error in all cout statements
2. Compiler error in last cout statements
3. 7
   7.0
   7.0
4. Runtime error

## STL

## Introduction

We have already understood the concept of C++ Template

The C++ STL (Standard Template Library) is a powerful set of C++ template classes to provide general-purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.

It is a generalized library and so, its components are parameterized.

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

1. Containers
2. Algorithms
3. Iterators

Learning STL is important for every C++ programmer as it saves a lot of time while writing code.

## Components

All the three components have a rich set of pre-defined functions which help us in doing complicated tasks in very easy fashion.

**Containers**
Containers are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, map etc.

**Algorithms**
Algorithms act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.

**Iterators**
Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers.

Will discuss about each component in detail soon

## Containers

## Containers

Containers are library used to manage collections of classes and objects of a certain kind.

The containers are implemented as generic class templates.

Containers help us to implement and replicate simple and complex data structures very easily like arrays, lists, trees , stack, queues, etc.

For example you can very easily define a linked list in a single statement by using list container of container library in STL , saving your time and effort. It means a linked list template is already defined. You have to simply use it by creating objects from it and calling methods of it.

Containers can be used to hold different kind of objects. It means same container can be operated on any data types , you don’t have to define the same container for different type of elements.

## Data structures

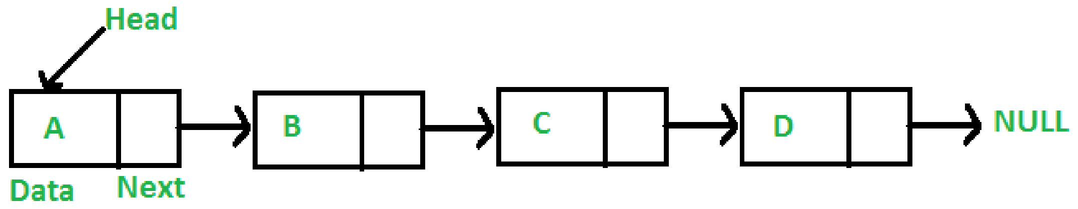
Array: is a linear collection of elements of similar data types. Operations possible on array : addition of elements. Addition can be done randomly.

Stack: collection of items arranged on top of each other in the form of pile where elements are inserted and extracted only from one end of the pile.. Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out)

Queue: A Queue is a linear structure which follows a particular order in which the operations are performed. specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other

## Data structures

Linked list:   A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations.

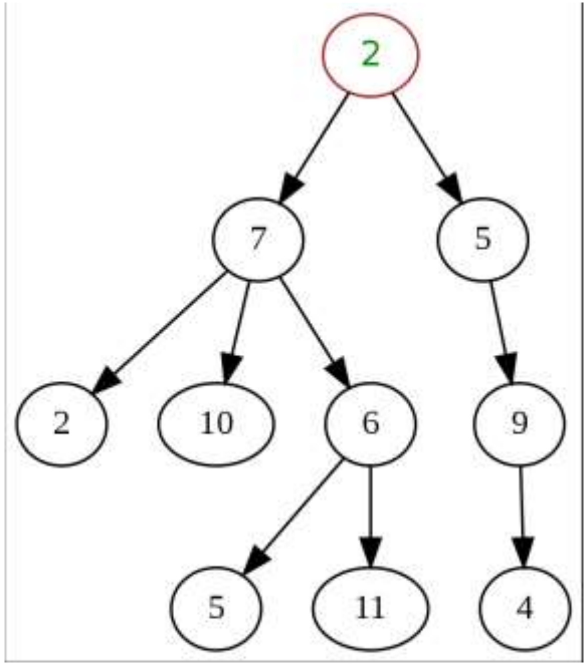


To create a linked list, following structure need to be created. Program will work around this structure.

```
// A linked list node
struct Node {
    int data;
    struct Node* next;
};
```

## Data structures

Tree: A tree is a nonlinear data structure, compared to arrays, linked lists, stacks and queues which are linear data structures. It is a collection of nodes connected by directed (or undirected) edges.



## Common containers

- vector: replicates arrays
- queue: replicates queue
- stack: replicates stack
- list: replicates linked list
- set: replicates trees
- maps: associative arrays
- And many more…

## How to use container library

When we use container library, then we have to include that header file first and use the constructor to initialize the object.

Eg. While using list container, include container list and create object as follows:

```
#include <iostream>
#include <list>

int main()
{
        list <int> mylist;
        list <double> mylist1;
        …..
        …..
}
```

## How to use container library

In the above example shown, we don’t have to create list class. It already exists.
There are pre-defined containers in c++, which are list, vector, queues , stacks , etc.

## Vector container in STL

- Earlier we have learnt array container in STL (or in general) using
  Array<int, 5> A;
- This array can contain 5 elements in an array named A. It is fixed size array.

- Drawbacks of array:
  - -The size of an array is fixed.
  - -User must know number of elements to be stored beforehand declaring an array.
  - - Defining oversized array is a wastage of memory ( to store 10 elements, we are declaring array of size)

- So we need solution which will allow us flexibility to add elements into an array as and when required at runtime

- Solution to this is Vector container in STL

## Vector container in STL

As we have identifie solution of the fixed size or static size arrays problem is dynamic arrays!

They have dynamic size, i.e. their size can change during runtime.

Container library provides vectors to replicate dynamic arrays.

SYNTAX for creating a vector is:

```
        vector< object_type > vector_name;
```

## Vector Container

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

int main() {
    vector <int> v;
}
```

Vector being a dynamic array, doesn't needs size during declaration, hence the above code will create a blank vector. Initially the vector is blank, as it has no data. but as you add data, it grows.

```
Vector <char> V1(5);

//it will create a vector V1 of size 5 initially which can grow dynamically
```

## Vector Container

1. There are many ways to initialize a vector

```
#include <iostream>
#include <vector>
using namespace std;
 int main() {
    vector<string> v {“c++" ,“STL" ,“looks" ,“great"};}
}
```

| | | | |
|---|---|---|---|
| C++ | STL | looks | great |

2. You can also initialize a vector with one element a certain number of times

```
vector<string> v(4 , “Test");
```

| | | | |
|---|---|---|---|
| Test | Test | Test | Test |

However this is not the end of the vector, still more elements can always  be added at the end.

## Member functions of vector

**push_back function:**
push_back() is used for inserting an element at the end of the vector. If the type of object passed as parameter in the push_back() is not same as that of the vector or is not interconvertible an exception is thrown.

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
   vector<int>  v; //will create a blank vector
   v.push_back(1);  //insert 1 at the back of v
   v.push_back(2);  //insert 2 at the back of v
   v.push_back(3);  //insert 3 at the back of v
}
```

## Member functions of vector

**Subscript Operator []**

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int>  v; //will create a blank vector
    v.push_back(1);  //insert 1 at the back of v
    v.push_back(2);  //insert 2 at the back of v
    v.push_back(3);  //insert 3 at the back of v
    cout<<v[0];    //prints 1
    cout<<v[1];    //prints 2
    cout<<v[2];  ///prints 3
}
```

## Member functions of vector

**Subscript Operator []**

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int>  v; //will create a blank vector
    v.push_back(1);  //insert 1 at the back of v
    v.push_back(2);  //insert 2 at the back of v
    v.push_back(3);  //insert 3 at the back of v
    for (int i=0;i<3;i++)
        {
            cout<<v[i];
        }
}
```

## Member functions of vector

- **size function** : This method returns the size of the vector.
- **empty function** :This method returns true if the vector is empty else returns false.
- **at function** : This method works same in case of vector as it works for array. vector_name.at(i) returns the element at **ith** index in the vector **vector_name**.
- **front and back functions** :vector_name.front() retuns the element at the front of the vector (i.e. leftmost element). While vector_name.back() returns the element at the back of the vector (i.e. rightmost element).
- **clear** function:This method clears the whole vector, removes all the elements from the vector but do not delete the vector. SYNTAX: clear() . For a vector **v**, v.clear() will clear it, but not delete it.
- **capacity() function:** This method returns the number of elements that can be inserted in the vector based on the memory allocated to the vector.

## Member functions of vector

```
int main()
   {
      vector<int>  v; //will create a blank vector
      cout<<"current capacity =" <<v.capacity()<<endl;
      for(int i=0;i<=9;i++)
      {
         v.push_back(10*(i+1));  //insert 10,20,30, upto 100 etc at the back of v
         cout<<"current capacity =" <<v.capacity()<<endl;
      }
      cout<<" Front element in vector " <<v.front()<<endl;
      cout<<" Back element in vector " <<v.back()<<endl;
```

## Member functions of vector

```
for (int i=0;i<v.size();i++)
        {
            cout<<v.at(i) <<" ";
        }
        v.clear();
        cout<<" \n size of vector"<<v.size()<<endl;
        if(v.empty())
            cout<<" Vector is empty " <<endl;
        cout<< " capacity of vector"<<v.capacity();
}
```

## Member functions of vector

```
Output:
current capacity =0
current capacity =1
current capacity =2
current capacity =4
current capacity =4
current capacity =8
current capacity =8
current capacity =8
current capacity =8
current capacity =16
current capacity =16
```

```
Front element in vector 10
Back element in vector 100
10 20 30 40 50 60 70 80 90 100
size of vector 0
Vector is empty
capacity of vector 16
```

## Size and capacity difference

Capacity and size are different functions. Size returns current number of elements where capacity function returns the size of the storage space currently allocated for the vector, expressed in terms of elements.

The vector::capacity() function is a built-in function which returns the size of the storage space currently allocated for the vector, expressed in terms of elements.

This capacity is not necessarily equal to the vector size.

It can be equal to or greater, with the extra space allowing to accommodate for growth without the need to reallocate on each insertion.

The capacity does not suppose a limit on the size of the vector.

## Size and capacity difference

```
using namespace std;

int main()
{
    vector<int> v;

    // inserts elements
    for (int i = 0; i < 10; i++) {
        v.push_back(i * 10);
    }

    cout << "The size of vector is " << v.size();
    cout << "\nThe maximum capacity is " << v.capacity();
    return 0;
}
```

## Size and capacity difference

```
Current size = 1 Current capacity allocated = 1
Current size = 2 Current capacity allocated = 2
Current size = 3 Current capacity allocated = 4
Current size = 4 Current capacity allocated = 4
Current size = 5 Current capacity allocated = 8
Current size = 6 Current capacity allocated = 8
Current size = 7 Current capacity allocated = 8
Current size = 8 Current capacity allocated = 8
Current size = 9 Current capacity allocated = 16
Current size = 10 Current capacity allocated = 16
The size of vector is 10
The maximum capacity is 16
```

## MCQ

What will the following line do?

```
vector <int> v3(5,10);
```

A. Create an integer vector v3 with 2 elements as 5,10.
B. Create an integer vector v3 of size 5 with every element value as 10
C. Compiler Reports an error as two values specified in vector size .
D. Compiler Reports an error as vector does not take size initially.

## MCQ

What will the following line do?

```
vector <int> v3(5,10);
```

A. Create an integer vector v3 with 2 elements as 5,10.
B. Create an integer vector v3 of size 5 with every element value as 10
C. Compiler Reports an error as two values specified in vector size .
D. Compiler Reports an error as vector does not take size initially.

## MCQ

Which of the following is not a function of Vector container in STL

1. at
2. empty
3. throw
4. size

## MCQ

Which of the following is not a function of Vector container in STL

1. at
2. empty
3. **throw**
4. size

// arr[i] == arr.at(i)

## Assignment

Create a vector of 5 strings and perform following functions and observe the output

1. Pop_back()
2. Pop_front()
3. Front()
4. Back()
5. Size()
6. Capacity()
7. Push_front()
8. Push_back()
9. At
10. empty

## QNA Time

**Any Questions??**

# Thank You!

**See you guys in next class.**