Chapter 6
Template and generic programming

Template

- Templates are the foundation of generic programming, which involves writing code in a way that is independent of any type.
- A template is a blueprint or formula for creating a generic class or a function.
- A template is one of the recently added feature in C++. It supports the generic data types and generic programming.
- Generic programming is an approach where generic data types are used as parameters in algorithms so that they can work for a variety of suitable data types.

  Example:

  i.   A class template for an **array class** would enable us to create arrays of various data types such as int array and float array.

  ii.   A function template say mul() can be used for multiplying int, float and double type values.

- A Template can be considered as Macro. When an object of specific type is defined for actual use, the template definition for that class is substituted with the required data type.
- Since a template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function, the template is also called as parametrized class or functions.

Features of Template

i.   Templates are easier to write. We can create only one generic version of our class or function instead of manually creating specializations.

ii.   Templates are easier to understand, since they provide a straightforward way of abstracting type information.

iii. Templates are type safe. Because the types that templates act upon are known at compile time, the compiler can perform type checking before errors occur.

Class Template

The general form of a generic class declaration is shown here:

> template<classgeneric_data_type>
>
> class class-name
>
> {
>
>   …………………..
>
> }

Advantage of class Template

i.   One C++ Class Template can handle different types of parameters.

ii.   Compiler generates classes for only the used types. If the template is instantiated for int type, compiler generates only an int version for the C++ template class.

iii. Templates reduce the effort on coding for different data types to a single set of code.

iv. Testing and debugging efforts are reduced.

Disadvantage of Template

i. Many compilers historically have very poor support for templates, so the use of templates can make code somewhat less portable.

ii. Almost all compilers produce confusing, unhelpful error messages when errors are detected in template code. This can make templates difficult to develop

iii. Each use of a template may cause the compiler to generate extra code (an instantiation of the template), so the indiscriminate use of templates can lead to code bloat, resulting in excessively large executable.

Example:

1. WAP to demonstrate the concept of generic class.

```
#include<iostream>
using namespace std;
template<class T>
class Demo
{
        private:
                T x;
        public:
                Demo( T p)
                {
                        x=p;
                }
                void show()
                {
                        cout<<"\n"<<x;
                }
};
int main()
{
        Demo <char *>p1("GCES");
        Demo <char> p2 ('A');
        Demo <int> p3(15);
        Demo <float>p4 (2.5);
        p1.show ();
        p2.show ();
        p3.show ();
        p4.show ();
```
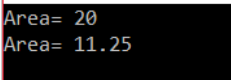
```
        }
```

Output:

        GCES

        A

        15

        2.5

2. WAP to find the area of rectangle using the concept of generic class.

```cpp
#include <iostream>
using namespace std;
template<class T> // here T is generic data type
class Rectangle
{
        private:
                T len,bre;
        public:
                Rectangle(T l, T b)
                {
                        len=l;
                        bre=b;
                }
                T area()
                {
                        return(len*bre);
                }
};
int main()
{
        Rectangle<int>r1(4,5);
        cout<<"Area= "<<r1.area()<<endl;
        Rectangle <float>r2(4.5, 2.5);
        cout<<"Area= "<<r2.area()<<endl;
}
```



```
Area= 20
Area= 11.25
```

In the above program, to define a class template Rectangle, we must prefix its definition by template<class T>. This prefix tells the compiler that we are going to declare a template and use T as a type name in the class definition. Thus, Rectangle has become parameterized class with T as its parameter. So, T can be substituted by any data type like int, float or any user defined data type.
Here in the example the statement Rectangle<int>r1(4,5); initialize T as int type and hence the class constructor receive integer type argument and method T area() also returns integer type value. Similarly, the statement Rectangle <float>r2(4.5, 2.5); initialize T as float type and hence the class constructor receive floating type argument and method T area() also returns Floating type value

Or , we can write the same program as below:

```cpp
#include <iostream>
using namespace std;
template<class T>  // here T is generic data type
class Rectangle
{
        private:
                T len,bre;
```

```cpp
                public:
                        setdata(T x, T y)
                        {
                                len=x;
                                bre=y;
                        }
                        T area()
                        {
                                return(len*bre);
                        }
};
int main()
{
        Rectangle<int> r1;
        r1.setdata(4,5);
        cout<<"Area= "<<r1.area()<<endl;
        Rectangle <float> r2;
        r2.setdata(4.5,2.5);
        cout<<"Area= "<<r2.area()<<endl;
}
```

3. WAP to find the sum of two complex number using the concept of generic class.

```cpp
#include <iostream>
using namespace std;
template<class T>
class complex
{
        T real;
        T img;
        public:
        complex()
        {
                real=0;
                img=0;
        }
        complex(T f1, T f2)
        {
                real=f1;
                img=f2;
        }

        complex sum(complex tmp)
        {
                complex result;
                result.real = tmp.real+real;
                result.img = tmp.img+img;
                return result;
        }


        void show()
        {
                cout<<"Real= "<<real<<endl;
                cout<<"Imaginary= "<<img<<endl;
        }
};

int main()
{
        //finding the sum of two integer complex number
```
4

```cpp
        complex<int> c1(3,6);
        complex<int> c2(2,-2);
        complex<int> c3;//will invoke default constructor
        c3=c1.sum(c2);
        c3.show();

        //finding the sum of two float complex number
        complex<float> c4(3.5,6.0);
        complex<float> c5(2.5,-2.5);
        complex<float> c6;//will invoke default constructor
        c6=c4.sum(c5);
        c6.show();
 }
```

```
Real= 5
Imaginary= 4
Real= 6
Imaginary= 3.5
```

4. WAP to find the sum of two complex number using the concept of generic class and operator overloading.

```cpp
#include <iostream>
using namespace std;
template<class T>
class complex
{
        T real;
        T img;
        public:
        complex()
        {
                real=0;
                img=0;
        }


        complex(T f1, T f2)
        {
                real=f1;
                img=f2;
        }

        complex operator +(complex tmp)
        {
                complex result;
                result.real = tmp.real+real;
                result.img = tmp.img+img;
                return result;
        }
        void show()
        {
                cout<<"Real= "<<real<<endl;
                cout<<"Imaginary= "<<img<<endl;
        }
};

int main()
{
        //finding the sum of two integer complex number
        complex<int> c1(3,6);
        complex<int> c2(2,-2);
        complex<int> c3;//will invoke default constructor
        c3=c1+c2; // equivalent to c3=c1.operator +(c2);
        c3.show();
```

```
                    //finding the sum of two float complex number
                    complex<float> c4(3.5,6.0);
                    complex<float> c5(2.5,-2.5);
                    complex<float> c6;//will invoke default constructor
                    c6=c4+c5; // equivalent to c6=c4.operator+(c5);
                    c6.show();
            }
```

```
Real= 5
Imaginary= 4
Real= 6
Imaginary= 3.5
```

Class Template with multiple parameters

When we have to use more than one generic data type in a class template, it can be achieved by using a comma-separated

list within the template specification as below.

```
        template<class T1, class T2………class TN>

        Class class_name

        {


        };
```

Example:

```
#include <iostream>
using namespace std;
template<class T1, class T2 >
class Demo
{
        private:
                T1 a;
                T2 b;
        public:
                Demo(T1 x, T2 y)
                {
                        a=x;
                        b=y;


                }
                void show()
                {
                        cout<<"A= "<<a<<endl;
                        cout<<"B= "<<b<<endl;
                }
};
int main()
{
        Demo<int,float> d1(1,4.5);
        d1.show();

        Demo<char,int> d2('B',6);
        d2.show();

        Demo <char *, float>d3("roman",6);
        d3.show();
}
```

Function Template

Like class, we can also define function templates that can be used to create a family of functions with different arguments types. The general format for function template is given below

> template<class T>
>
> returnType of type Tfunction_name (argument of type T)
>
> {
>
> //body of function with type T whenever appropriate
>
> }

Note: Main function can't be declared as template

Example:

1. WAP to declare template function that can be used to find the area of rectangle.

```
#include<iostream>
using namespace std;
template<class t>
t area(t len, t bre)
{
        return(len*bre);
}
int main()
{
        int l1=6,b1=4;
        cout<<"Area= "<<area(l1,b1)<<endl;
        float l2=2.5,b2=2.0;
        cout<<"Area= "<<area(l2,b2);
}
```

2. WAP to declare a function template that can be used to swap two values of a given type of data.

```
//using concept of pointer
#include <iostream>
using namespace std;
template<class T >
void swap( T *fn, T *sn )
{
        T tmp;
        tmp=*fn;
        *fn=*sn;
        *sn=tmp;
}

int main()
{
        int m=5,n=6;
        cout<<"Before swapping"<<endl;
        cout<<"M= "<<m<<endl<<"N= "<<n<<endl;
        swap(&m,&n);
        cout<<"After swapping"<<endl;
        cout<<"M= "<<m<<endl<<"N= "<<n<<endl;
}
```

Or

```
//Using concept of reference variable
#include <iostream>
using namespace std;
template<class T >
void swap( T &fn, T &sn )
```

7

```cpp
{
    T tmp;
    tmp=fn;
    fn=_sn;
    fn=tmp;
}

int main()
{
    int m=5,n=6;
    cout<<"Before swapping";
    cout<<"M= "<<m<<endl<<"N= "<<n<<endl;
    swap(m,n);
    cout<<"After swapping";
    cout<<"M= "<<m<<endl<<"N= "<<n<<endl;
}
```

3. Create a function template that can be used to find the maximum numbers among 10 numbers of given type of data(int, float etc)

```cpp
#include <iostream>
using namespace std;
template<class T >
T Max( T data[] )
{
    T great=data[0];
    for(inti=0;i<10;i++)
    {
        if(data[i]>great)
        great=data[i];
    }
    return great;
}

int main()
{
    int a[10]={4,3,7,9,8,5,88,34,23,11};
    cout<<"Largest number= "<<Max(a)<<endl;

    float b[10]={4.5,3.6,7.6,99.8,8,5,88.8,34.3,23.5,11.5};
    cout<<"Largest number= "<<Max(b);
}
```

```
Largest number= 88
Largest number= 99.8
```

4. Create a function templates that can be used to find the minimum and maximum numbers among 10 numbers of given type of data(int, float etc)

```cpp
#include <iostream>
using namespace std;
template<class T >
T Max( T data[] )
{
    T great=data[0];
    for(inti=0;i<10;i++)
    {
        if(data[i]>great)
        great=data[i];
    }
    return great;
}

template<class T >
```

8

```cpp
T Min( T data[] )
{
        T small=data[0];
        for(inti=0;i<10;i++)
        {
                if(data[i]<small)
                small=data[i];
        }
        return small;
}

int main()
{
        int a[10]={4,3,7,9,8,5,88,34,23,11};
        cout<<"Largest number= "<<Max(a)<<endl;
        cout<<"Smallest number= "<<Min(a)<<endl;

        float b[10]={4.5,2.6,7.6,99.8,8,5,88.8,34.3,23.5,11.5};
        cout<<"Largest number= "<<Max(b)<<endl;
        cout<<"Smallest number= "<<Min(b)<<endl;
}
```

```
Largest number= 88
Smallest number= 3
Largest number= 99.8
Smallest number= 2.6
```

5. Create a function template to add two matrices.

```cpp
#include<iostream>
using namespace std;
template<class t>
void add(t x [][10], t y [][10], t row, t col)
{
        tsm[10][10];
        for(inti=0;i<row;i++)
        {
                for(int j=0;j<col;j++)
                {
                        sm[i][j]=x[i][j]+y[i][j];
                }
        }
        for(inti=0;i<row;i++)
        {
                for(int j=0;j<col;j++)
                {
                        cout<<sm[i][j]<<" ";
                }
                cout<<endl;
        }
}

int main()
{
        int m[10][10],n[10][10],row,col;
        cout<<"Enter the number of rows and column"<<endl;
        cin>>row>>col;
        cout<<"Enter the elements of first matrix"<<endl;
        for(inti=0;i<row;i++)
        {
                for(int j=0;j<col;j++)
                {
                        cin>>m[i][j];
                }
        }
```

```
                cout<<"Enter the elements of second matrix"<<endl;
                for(inti=0;i<row;i++)
                {
                        for(int j=0;j<col;j++)
                        {
                                cin>>n[i][j];
                        }
                }
                add(m,n,row,col);
        }
```

Function template with multiple parameters

Like template class, we can use more than one generic data type in the template statement, using a comma separated list as shown below.

```
        template<class t2, class t2………>

        Return_typefunction_name(arguments of types T1, T2……..)

        {


        }
```

Example:

```
        #include <iostream>
        using namespace std;
        template<class T1, class T2 >
        void show(T1 x, T2 y)
        {
                cout<<"First Parameter= "<<x<<endl;
                cout<<"Second Paramter= "<<y<<endl;
        }
        int main()
        {
                show(1,2);
                show(1.4,3.5);
                show('r',"Ram");
        }
```

```
First Parameter= 1
Second Paramter= 2
First Parameter= 1.4
Second Paramter= 3.5
First Parameter= r
Second Paramter= Ram
```

Overloading of Template Function

A template function can be overloaded either by template functions or ordinary functions of its name.In such case, overloading resolution is accomplished as follows

   i.   Calling an ordinary function that has an exact match.
   ii.   Calling a template function that could be created with an exact match.
   iii.   Trying normal overloading resolution to ordinary functions and call the one that matches.

An error is generated if no match is found. Also no automatic conversion are applied to arguments on the template function.

Example:

```
        #include<iostream>
        using namespace std;
        template<class t>
        void display(t x)
```

10

```cpp
{
        cout<<"From Template Function X= "<<x<<endl;
}


template<class  t1, class  t2>
void display(t1 x, t2 y)
{
        cout<<"From Template Function"<<endl<<"X= "<<x<<endl<<"Y= "<<y<<endl;
}
void display(float x)
{
        cout<<"From Ordinary Function X= "<<x<<endl;
}
void display(float x, float y)
{
        cout<<"From Ordinary Function"<<endl<< "X= "<<x<<endl<<"Y= "<<y<<endl;
}

int main()
{
        display(100);//will invoke template function with one argument as no exact matching ordinary function exist.
        display(100.5f);// will invoke ordinary function as exact matching ordinary function exist
        display(2.4f,3.5f);//will invoke ordinary function as exact matching ordinary function exist
        display(3,5);//will invoke template function with two argument because non ordinary function matches this signature
        }
```

```
From Template Function X= 100
From Ordinary Function X= 100.5
From Ordinary Function
X= 2.4
Y= 3.5
From Template Function
X= 3
Y= 5
```

Example 2:

```cpp
#include<iostream>
using namespace std;
void display(float x)
{
        cout<<"From Ordinary Function1 X= "<<x<<endl;
}

template<class t1, class t2>
void display (t1 x, t2 y)
{
        cout<<"From template Function X= "<<x<<" Y= "<<y<<endl;
}


int main()
{
        display(100); //will invoke ordinary function1 with one argument of type float(i.e. int casted to
float)
        display("bhesh",5);
}
```

```
From Ordinary Function1 X= 100
From template Function X= bhesh Y= 5
```

Exception Handling in C++

Introduction to error

In computing, an error in a program is due to the code that does not conform to the order expected by the programming language. An error may produce and incorrect output or may terminate the execution of program abruptly or even may cause the system to crash.

Types of Errors
    i.   Compile Time Error
    ii.   Runtime Error

Compile Time Error

At compile time, when the code does not comply with the C++ syntactic and semantics rules, compile-time errors will occur. The goal of the compiler is to ensure the code is compliant with these rules. Any rule-violations detected at this stage are reported as compilation errors. These errors are checked.

The following are some common compile time errors:

- Writing any statement with incorrect syntax
- Using keyword as variable name
- Attempt to refer to a variable that is not in the scope of the current block
- A class tries to reference a private member of another class
- Trying to change the value of an already initialized constant (final member)
  etc……….

Runtime Error

When the code compiles without any error, there is still chance that the code will fail at run time. The errors only occur at run time are call run time errors. Run time errors are those that passed compiler's checking, but fail when the code gets executed. These errors are unchecked.

The following are some common runtime errors:

- Divide by zero exception
- Array index out or range exception
- StackOverflow exception
- Dereferencing of an invalid pointer
  etc……….

So, runtime errors are those which are generally can't be handled and usually refers catastrophic failure.
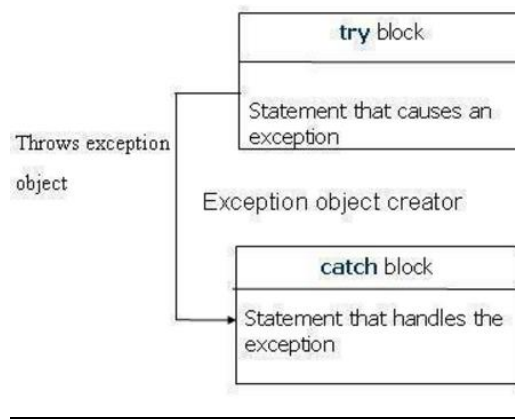
Exception

An exception is a run-time error. Proper handling of exceptions is an important programming issue. This is because exceptions can and do happen in practice and programs are generally expected to behave gracefully in face of such exceptions.

Unless an exception is properly handled, it is likely to result in abnormal program termination and potential loss of work. For example, an undetected division by zero or dereferencing of an invalid pointer will almost certainly terminate the program abruptly.

Types

1. **Synchronous:** Exception that are caused by events that can be control of program is called synchronous exception. Example, array index out or range exception.
2. **Asynchronous:** Exception that are caused by events beyond the control of program is called asynchronous exception. Example, Exception generated by hardware malfunction.

Basic steps in exception handling

The **purpose/Use** of exception handling mechanism is to provide means to detect and report an exception so that appropriate actions can be taken. Exception handling mechanism in C++ consists of four things:

    i.   Detecting of a run-time error (Hit the exception)
    ii.   Raising an exception in response to the error (Throw the exception)
    iii.   Receive the exception information (Catch the exception)
    iv.   Taking corrective action. (Handle the exception)

C++ provides a language facility for the uniform handling of exceptions. Under this scheme, a section of code whose execution may lead to run-time errors is labeled as a **try block**. Any fragment of code activated during the execution of a try block can raise an exception using a **throw clause**. All exceptions are typed(i.e., each exception is denoted by an object of a specific type). A try block is followed by one or **more catch clauses**. Each catch clause is responsible for the handling of exceptions of a particular type. When an exception is raised, its type is compared against the catch clauses following it. If a matching clause is found, then its handler is executed. Otherwise, the exception is propagated up, to an immediately enclosing try block (if any). The process is repeated until either the exception is handled by a matching catch clause or it is handled by a default handler.

The general form is as below:

```
..........
..........
try
{
        .....
        .....
        throw exception; //block of statements which detect and throws an exception
}
catch(type arg)   //catches exception
{
        .....
        ....   //block of statements that handle the exception
        .....
}
.......
.......
```

Example:
WAP to input two number and divide first number by second. The program must handle the divide by zero exception.

```cpp
#include<iostream>
using namespace std;
int main()
{
        int nu,de,res;
        cout<<"Enter numerator and denominator";
        cin>>nu>>de;
        try
        {
                if(de==0)
                {
                        throw(de); //throw int object
                }
                else
                {
                        res=nu/de;
                        cout<<"Result= "<<res;
                }
        }
        catch(inti)          //catches the int type exception
        {
                cout<<"Divide by zero exception occurred: de= "<<i;
        }
}
```

```
Enter numerator and denominator4
2
Result= 2
```
```
Enter numerator and denominator4
0
Divide by zero exception occured: de= 0
```

So, in the above program, when no exception is thrown, the catch block is skipped and outputs correct result. But when the user input zero for denominator, the exception is thrown using throw statement with int type object as argument. Since the exception object type is int, the exception handler i.e. catch statement containing int type argument catches the exception and handles it by displaying necessary message or performing necessary steps further.

Note:
   i.   During execution, when throw statement is encountered, then it immediately transfers the control suitable exception handler i.e. catch block. Hence all the statement after throw in try block are skipped.
   ii.   If there are multiple catch block with integer type argument, then the first one immediately after try block gets executed.
   iii.   When an exception object is thrown and non of the catch block matches, the program is aborted with the help of abort() function which is invoked automatically.

Nesting try statement
A try block can be placed inside the block of another try, called as nested try.

Example:
WAP to input two numbers and divide first number by second. The result must be stored in the index entered by user.

```cpp
#include<iostream>
using namespace std;
int divide(int,int);




int main()
{
```

```
        inta,b,ind,res;
        int c[10];
        cout<<"Enter numerator and denominator"<<endl;
        cin>>a>>b;
        try
        {
                if(b==0)
                {
                        throw(b);
                }
                cout<<"Enter the index to store the result";
                cin>>ind;
                try
                {
                        if(ind>=10)
                        {
                                throw(ind);
                        }
                        res=a/b;
                        c[ind]=res;
                        cout<<"Result "<<res<<" successfully stored at index "<<ind;
                }
                catch(int e)
                {
                        cout<<"Index out of range exception occurred: index= "<<ind;
                }
        }
        catch(int e )
        {
                cout<<"Divide by zero exception occured: b="<<b;
        }
}
```

```
Enter numerator and denominator
4
2
Enter the index to store the result8
Result 2 successfully stored at index 8
Enter numerator and denominator
4
0
Divide by zero exception occured: b=0
Enter numerator and denominator
4
2
Enter the index to store the result88
Index out of range exceptoin occured: index= 88
```

Multiple catch statement

It is also possible that a program segment has more than one condition to throw an exception. In such cases, we can associate more than one catch statement with a try as shown below.

```
        try
        {
                catch(type1object);
                catch(type2object);
                …………………
                catch(typeNobject);
        }
        catch(type1 arg)
        {

        }
```

15

```cpp
        catch(type2 arg)
        {

        }
        catch(typeNarg)
        {

        }
```

Example:

```cpp
        #include<iostream>
        using namespace std;
        int divide(int,int);
        int main()
        {
                try
                {
                        double nu,de,res[10],c;
                        int ind;
                        cout<<"Enter numerator and denominator"<<endl;
                        cin>>nu>>de;
                        if(de==0)
                        {
                                throw(de);
                        }
                        cout<<"Enter the index to store the result"<<endl;
                        cin>>ind;
                        if(ind>=10)
                        {
                                throw(ind);
                        }
                        c=nu/de;
                        res[ind]=c;
                        cout<<"Result= "<<c<<" Successfully stored at "<<ind<<" posiion";
                }
                catch(int a)
                {
                        cout<<"Index out or range exception: Index= "<<a;
                }
                catch(double d)
                {
                        cout<<"Divide by zero exception: Denominator= "<<d;
                }
        }
```

```
Enter numerator and denominator
4
2
Enter the index to store the result
6
Result= 2 Successfully stored at 6 position
```

```
Enter numerator and denominator
4
0
Divide by zero exception: Denominator= 0
--------------------------------
```

```
Enter numerator and denominator
4
3
Enter the index to store the result
88
Index out or range exception: Index= 88
```

Standard Template Library

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.

STL has three components
- Algorithms
- Containers
- Iterators
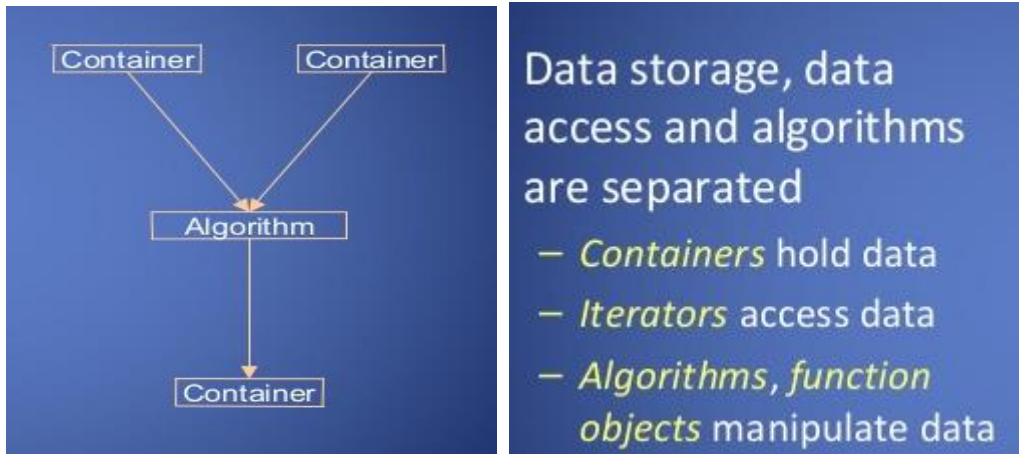
Algorithms
The header algorithm defines a collection of functions especially designed to be used on ranges of elements. They act on containers and provide means for various operations for the contents of the containers.
- Algorithm
    - Sorting
    - Searching
    - Important STL Algorithms
    - Useful Array algorithms
    - Partition Operations

Containers
Containers or container classes store objects and data. There are in total seven standard "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 which can be accessed in a sequential manner.
    - vector
    - list
    - deque
    - arrays
    - forward_list
- Container Adaptors: provide a different interface for sequential containers.
    - queue
    - priority_queue
    - stack
- Associative Containers : implement sorted data structures that can be quickly searched
    - set
    - multiset
    - map
    - multimap

Iterators

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