UNIT-V

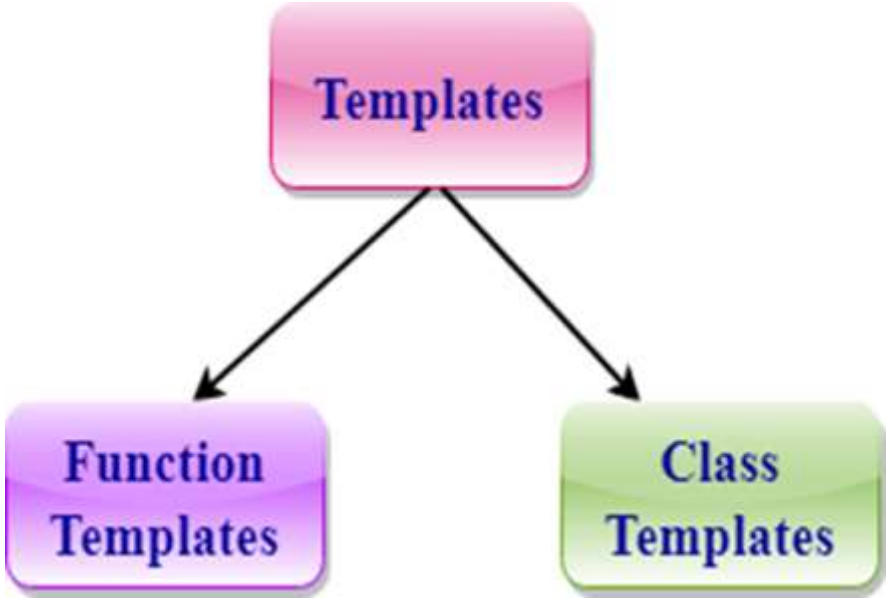
Generic Programming with Templates & Exception Handling: Definition of class Templates, Normal Function Templates, Over Loading of Template Function, Bubble Sort Using Function Templates, Difference between Templates and Macros, Linked Lists with Templates,

Exception Handling: Principles of Exception Handling, The Keywords try throw and catch, Multiple Catch Statements, Specifying Exceptions.

Overview of Standard Template Library: STL Programming Model, Containers, Sequence Containers, Associative Containers, Algorithms, Iterators, Vectors, Lists, Maps.

Template

- 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 can define a template for a function. 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.

- 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.
- A single function template **can work with different data types at once** but, a single normal function can only work with one set of data types.
- The type of the data that the function will operate on depends on the type of the data passed as a parameter.
- However, a better approach would be to use function templates because you can perform the same task writing less and maintainable code.

How to declare a function template?

```
template <class T>
T someFunction(T arg)
{
   ... .. ...
}
```

In the above code, T is a template argument that accepts different data types (int, float), and class is a keyword.

When, an argument of a data type is passed to someFunction( ), compiler generates a new version of someFunction() for the given data type.

Example program

```
#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;
}
```

Function Templates with Multiple Parameters

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

Syntax

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

return_type function_name (arguments of type T1, T2....)

{

    // body of function.

}
```

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

Example program

```
#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.

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.

Sometimes, you need a class implementation that is same for all classes, only the data types used are different.

class templates make it easy to reuse the same code for all data types

How to declare a class template?

```
template <class T>
class className
{
  … .. …
public:
  T var;
  T someOperation(T arg);
  … .. …
};
```

Here, T is the template argument which is a placeholder for the data type used.

Inside the class body, a member variable var and a member function someOperation() are both of type T.

How to create a class template object?

To create a class template object, you need to define the data type inside a < > when creation.

className<dataType>classObject;

className classObject; // normal class object

For example:

1. className<int> classObject;
2. className<float> classObject;
3. className<string> classObject;

Example program

```
#include <iostream>
using namespace std;
template<class T>
class A
{
   public:
   T num1 = 5;
   T num2 = 6;
   void add()
   {
      cout << "Addition of num1 and num2 : " << num1+num2<<endl;
   }
};
int main()
{
    A<int> d;
     d.add();
    return 0;
}
```

CLASS TEMPLATE WITH MULTIPLE PARAMETERS

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

Syntax

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

class class_name

{

   // Body of the class.

}
```

Example program

```
#include<iostream>
using namespace std;
template<class T>
class data
{
        public:
                T a;
                T b;
        public:
                data(T x,T y)
                {
                        a=x;
                        b=y;
                }
                void display()
                {
                        cout<<a<<b;
                }
};
```

```
int main()
{
        data<int>obj1(10,20);
        obj1.display();
        data<float>obj2(10.2,30.5);
        obj2.display();
}
```

2. 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
{
   private:
                    T1 a;
                    T2 b;
   public:
          A(T1 x,T2 y)
          {
                    a = x;
                    b = y;
      }
      void display()
      {
          cout<<"Value of a:"<<a<<endl;
          cout<<"Value of b:"<<b<<endl;
      }
};
int main()
{
   A<int,float>d(5,6.5);
   d.display();
}
```

Output

```
Value of a:5
Value of b:6.5
```

Template Function Overloading:

- The name of the function templates are the same but called with different arguments is known as function template overloading.
- If the function template is with the ordinary template, the name of the function remains the same but the number of parameters differs.
- When a function template is overloaded with a non-template function, the function name remains the same but the function's arguments are unlike.

Template Function Overloading:

program to illustrate overloading of template function using an explicit function

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

// Template declaration
template <class T>

// Template overloading of function
void display(T t1)
{
   cout << "Displaying Template: "<< t1 << "\n";
}
// Template overloading of function
void display(int t1)
{
   cout << "Explicitly display: " << t1 << "\n";
}
int main()
{
   // Function Call with a different arguments
   display(200);
   display(12.40);
   display('G');
   return 0;
}
```

Output:

```
Explicitly display: 200

Displaying Template: 12.4

Displaying Template: G
```

Macros

- A **macro** is a piece of code in a program that is replaced by the value of the macro.
- Macro is defined by **#define** directive. Whenever a macro name is encountered by the compiler, it replaces the name with the definition of the macro.
- Macro definitions need not be terminated by semi-colon(**;**).

```
#define LIMIT 5
int main()
{
   // Print the value of macro defined
   printf("The value of LIMIT is %d", LIMIT);
    return 0;
}
```

Output:
The value of LIMIT is 5

```
// Macro definition
#define AREA(l, b) (l * b)
 int main()
{
   // Given lengths l1 and l2
   int l1 = 10, l2 = 5, area;
    // Find the area using macros
    area = AREA(l1, l2);
    // Print the area
   printf("Area of rectangle is: %d", area);
    return 0;
}
```

How does template differ from macro?

S.No	Macro	Template
1	Macro are used in C and C++ for replacement of numbers	Template is only available in C++ language. It is used to write small macro like functions etc
2	Macro cannot check the data type of arguments	Template checks the data type
3	The code generated by the Macro can take a certain time to process.	templates will always take longer to process
4	Macros will not	templates are significantly more powerful and obey C++ syntactical rules
5	Macros cannot work recursively	Templates able to work recursively
6	Macros are pure text substitution.	Templates are for datatype substitution.

Exceptional handling

- It is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors.
- In C++, exception is an event or object which is thrown at runtime.
- All exceptions are derived from **std::exception** class.
- It is a runtime error which can be handled.
- If we don't handle the exception, it prints exception message and terminates the program.

Advantage:

- It maintains the normal flow of the application. In such case, rest of the code is executed even after exception.

Exceptional handling

In C++, we use **3 keywords** to perform exception handling:

- try
- catch, and
- throw

Syntax:

```
try
{
   //code
   throw parameter;
}
catch(exceptionname ex)
{
   //code to handle exception
}
```

- **try block**
- The code which can throw any exception is kept inside(or enclosed in) a try block.
- Then, when the code will lead to any error, that error/exception will get caught inside the catch block.
- **catch block**
- catch block is intended to catch the error and handle the exception condition. We can have multiple catch blocks to handle different types of exception and perform different actions when the exceptions occur.
- For example, we can display descriptive messages to explain why any particular excpetion occurred
- **throw statement**
- It is used to throw exceptions to exception handler i.e. it is used to communicate information about error. A throw expression accepts one parameter and that parameter is passed to handler.
- throw statement is used when we explicitly want an exception to occur, then we can use throw statement to throw or generate that exception.

Understanding Need of Exception Handling

- In C++ programming, exception handling is performed using try/catch statement.
- The C++ **try block** is used to place the code that may occur exception.
- The **catch block** is used to handle the exception.

C++ example without try/catch

```
#include <iostream>
using namespace std;
float division(int x, int y)
 {
   return (x/y);
}
int main ()
{
   int i = 50;
   int j = 0;
   float k = 0;
   k = division(i, j);
   cout << k << endl;
   return 0;
}
```

Output:

Floating point exception (core dumped)

C++ try/catch example

```
#include <iostream>

using namespace std;

float division(int x, int y)

{

  if( y == 0 )

  {

    throw "Attempted to divide by zero!";

  }

  return (x/y);

}
```

```
int main ()
{
  int i = 25;
  int j = 0;
  float k = 0;
  try
  {
    k = division(i, j);
    cout << k << endl;
  }
catch (const char* e)
 {
    cerr << e << endl;
 }
  return 0;
}
```

Output:

Attempted to divide by zero!

Principles of Exception Handling

1.If you can't handle an exception, don't catch it.

2.Catch an exception as close as possible to its source.

3.If you catch an exception, don't swallow it.

4.Log an exception where you catch it, unless you plan to re-throw it.

Multiple Catch Statements

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 (similar to switch statement). The format of multiple catch statements is as follows:

```
try
{
// try block
}
catch (type1 arg)
{
// catch section1
}
catch (type2 arg)
{
// catch section2
}
.......
.......
catch (typen arg)
{
// catch section-n
}
```

Multiple Catch Statements

```
#include <iostream>
#include<conio.h>
using namespace std;
int main()
{
   int x[3] = {-1,2};
   for(int i=0; i<2; i++)
   {
      int ex =x[i];
      try
      {
         if (ex < 0)
            // throwing numeric value as exception
            throw ex;
         else
               // throwing a character as exception
               throw “ex”;
         }
      catch (int ex)  // to catch numeric exceptions
         {
            cout << "Integer exception\n";
         }
         catch (char ex) // to catch character/string
exceptions
         {
            cout << "Character exception\n";
         }
   }
}
```

Output:

Integer exception

Character exception

Standard Exceptions in C++

There are some standard exceptions in C++ under <exception> which we can use in our programs. They are arranged in a parent-child class hierarchy which is depicted below:

std::exception - Parent class of all the standard C++ exceptions.
logic_error - Exception happens in the internal logical of a program.
domain_error - Exception due to use of invalid domain.
invalid argument - Exception due to invalid argument.
out_of_range - Exception due to out of range i.e. size requirement exceeds allocation.
length_error - Exception due to length error.
runtime_error - Exception happens during runtime.
range_error - Exception due to range errors in internal computations.
overflow_error - Exception due to arithmetic overflow errors.
underflow_error - Exception due to arithmetic underflow errors
bad_alloc - Exception happens when memory allocation with new() fails.
bad_cast - Exception happens when dynamic cast fails.
bad_exception - Exception is specially designed to be listed in the dynamic-exception-specifier.
bad_typeid - Exception thrown by typeid.

Specifying Exceptions

- It is possible to restrict a function to throw only certain specified exceptions.
- This is achieved by adding a throw list clause to the function definition. The general form of using an exception specification is:

```
return_type function_name (parameter list) throw (data type list)
{
// function body
}
```

- The data type list indicates the type of exception that is permitted to be thrown.
- If we want to deny a function from throwing any exception, declaring the data type list void as per the following statement can do this.

```
throw(); // void or vacant list
```

/* Write a C++ program to restrict a function to throw only specified type of exceptions. */

```
#include<iostream>
using namespace std;
void check (int k) throw (int)
{
if (k==1) throw 'k';
else
if (k==2) throw k;
else
if (k==-2) throw 1.0;
}
int main()
{
try
{
check(1);
check(-2);
check(3);
}
catch (char g)
{
cout<<"Caught a character exception \n";
}
catch (int j)
{
cout<<"Caught a character exception \n";
}
catch (double s)
{
cout<<"Caught a double exception \n";
}
cout<<"\n End of main()";
return 0;
}
```

STL Programming Model

- 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.
- STL has four components
    1. Algorithms
    2. Containers
    3. Functions
    4. 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.

Points to Remember:

- Algorithms provide approx **60 algorithm functions** to perform the complex operations.
- Standard algorithms allow us to work with two different types of the container at the same time.
- Algorithms are not the member functions of a container, but they are the standalone template functions.
- **Algorithms save a lot of time and effort.**
- If we want to access the STL algorithms, we must include the **<algorithm>** header file in our program.

STL algorithms can be categorized as:

Non mutating algorithms: Non mutating algorithms are the algorithms that **do not alter any value** of a container object nor do they change the order of the elements in which they appear. These algorithms can be used for all the container objects, and they make use of the forward iterators.

Mutating algorithms: Mutating algorithms are the algorithms that can be **used to alter the value** of a container. They can also be used to change the order of the elements in which they appear.

Sorting algorithms: Sorting algorithms are the modifying algorithms **used to sort the elements** in a container.

Set algorithms: Set algorithms are also known as sorted range algorithm. This algorithm is used to perform some function on a container that greatly **improves the efficiency of a program.**

Relational algorithms: Relational algorithms are the algorithms used to work on the **numerical data**. They are mainly designed to perform the **mathematical operations** to all the elements in a container.

Containers

- Containers or container classes store objects and data.
- Each container class contains a set of functions that can be used to manipulate the contents.

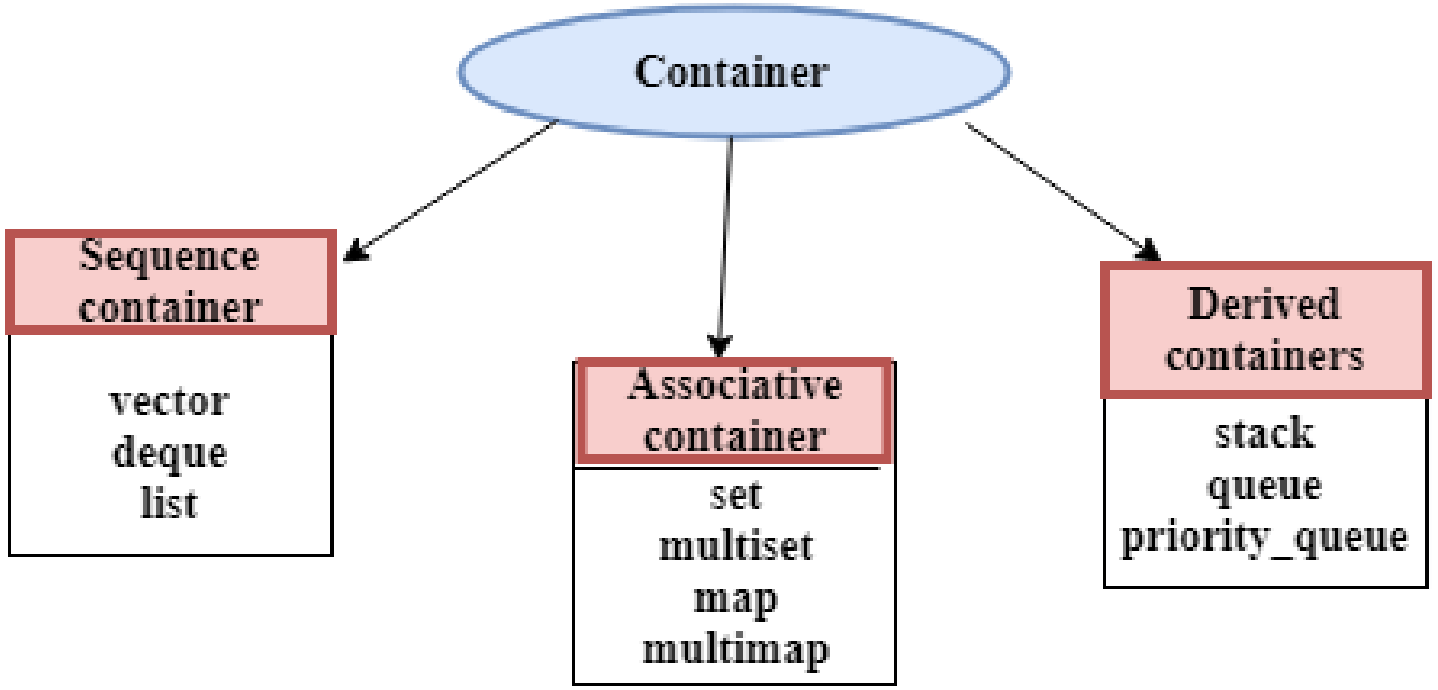
Sequence Containers:

implement data structures which

can be accessed in a sequential

manner.

- vector
- list
- deque
- arrays



Associative Containers

Associative containers **implement sorted data structures** that can be quickly searched (O(log n) complexity).

Set: Collection of unique keys, sorted by keys (class template)

Map: Collection of key-value pairs, sorted by keys, keys are unique (class template).

multiset: Collection of keys, sorted by keys (class template)

multimap: Collection of key-value pairs, sorted by keys (class template)

ITERATOR

- Iterators are pointer-like entities used to access the individual elements in a container.
- Iterators are moved sequentially from one element to another element. This process is known as **iterating through a container.**

Important Points:

- Iterators are used to traverse from one element to another element, a process is known as **iterating through the container**.
- The main advantage of an iterator is to provide a common interface for all the containers type.
- Iterators make the **algorithm independent** of the type of the container used.
- Iterators provide a generic approach to navigate through the elements of a container.

Syntax:

```
<containerType> :: iterator;
<containerType> :: const_iterator;
```

Operations Performed on the Iterators:

- **Operator (*) :** The '*' operator returns the element of the current position pointed by the iterator.
- **Operator (++) :** The '++' operator increments the iterator by one. Therefore, an iterator points to the next element of the container.
- **Operator (==)** and **Operator (!=) :** Both these operators determine whether the two iterators point to the same position or not.
- **Operator (=) :** The '=' operator assigns the iterator.

Vectors

- A vector is a sequence container class that implements dynamic array, means size automatically changes when appending elements. A vector stores the elements in contiguous memory locations and allocates the memory as needed at run time.

Difference between vector and array

- An array follows static approach, means its size cannot be changed during run time while vector implements dynamic array means it automatically resizes itself when appending elements.

Syntax

Consider a vector 'v1'. Syntax would be:

vector<object_type> v1;

Example

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
        vector<string> v1;
        v1.push_back("javaTpoint ");
        v1.push_back("tutorial");
        for(vector<string>::iterator itr=v1.begin();itr!=v1.end();++itr)
        cout<<*itr;
        return 0;
}
```

Output:

javaTpoint tutorial

Lists

- List is a contiguous container while vector is a non-contiguous container i.e list stores the elements on a contiguous memory and vector stores on a non-contiguous memory.
- Insertion and deletion in the middle of the vector is very costly as it takes lot of time in shifting all the elements. Linklist overcome this problem and it is implemented using list container.
- List supports a bidirectional and provides an efficient way for insertion and deletion operations.
- Traversal is slow in list as list elements are accessed sequentially while vector supports a random access.

Template for list

```
#include<iostream>
#include<list>
using namespace std;
int main()
{
  list<int> l;
}
```

It creates an empty list of integer type values.

List can also be initalised with the parameters.

```
#include<iostream>
#include<list>
using namespace std;
int main()
{
  list<int> l{1,2,3,4};
}
```

List can be initialised in two ways.

```
list<int>  new_list{1,2,3,4};
or
list<int> new_list = {1,2,3,4};
```

Maps

Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have the same key values.

Keys	Values
101	Nikita
102	Robin
103	Deep
104	John

Maps

Syntax

```
template < class Key,                          // map::key_type
        class T,                               // map::mapped_type
        class Compare = less<Key>,   // map::key_compare
        class Alloc = allocator<pair<const Key,T> >    // map::allocator_type
        > class map;
```

key: The key data type to be stored in the map.

type: The data type of value to be stored in the map.

compare: A comparison class that takes two arguments of the same type bool and returns a value. This argument is optional and the binary predicate less<"key"> is the default value.

alloc: Type of the allocator object. This argument is optional and the default value is allocator .

Creating a map

Maps can easily be created using the following statement:

map<key_type , value_type> map_name;

The above form will use to create a map with key of type **Key type** and value of type **value type.** One important thing is that key of a map and corresponding values are always inserted as a pair, you cannot insert only key or just a value in a map.