6. TEMPLATE AND GENERIC PROGRAMMING

Template

- Template is a new concept which enable us to define generic classes and functions and thus provides support for generic programming.
- Generic programming is an approach where generic types are used as parameters so that they work for variety of suitable data types and data structures.
- A template can be used to create a family of classes or functions. For example, a class template for an array class would enable us to create arrays of various data types such as int array, float array, char array.
- Similarly, we can define a function template that would create various versions of that function for different data types like int, float, double, char etc.
- Template types are:
    - ➢Function Template
    - ➢Class Template

1. Function Template

- We can define function templates that could be used to create a family of functions with different argument types. The general format of function template is:

```
template <class T>
return_type function_name(argument of type T)
{
    //...
}
```

- The function template syntax is like that of the class template except that we are defining functions instead of classes. We must use the template parameter T as parameter and when necessary in the function body and in its argument list.

//program51, program52, program53

Function Templates With Multiple Parameters

We can use more than one generic data type in the template statement as below:

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

//program54

2. Class Template

```
class Vector
{
private:
        int *v;
        int size;
public:
        Vector(int m)
        {
        size = m;
        v = new int[size];
        for(int i = 0; i < size;
i++)//null vector
                v[i] = 0;
        }
        Vector(int *a)
        {
        for(int i = 0; i < size;i++)
                v[i]=a[i];

        }
```

```
        int operator*(Vector &x)
        {
        int sum=0;
        for(int i=0;i<size;i++)
                sum+=v[i]+x.v[i];
        return sum;
        }

};
```

Class Template cont.

- The vector class can store an array of int numbers and perform the scalar product of two int vectors as shown below:

```
int main()
{
  int x[]={10,20,30};
  int y[]={1,2,3};
  Vector v1(3);
  Vector v2(3);
  v1=x;
  v2=y;
  int result=v1*v2;
  cout<<"Resultant magnitude: "<<result<<endl;
  return 0;
}
```

- Now, suppose we want to define a vector that can store an array of float values. We can do it by simply replacing the int with float in the vector class. This means that we have to redefine the entire class all over again.

Class Template cont.

- Template allows us to define generic classes. It is a simple process to create a generic class using a template with an anonymous/undefined type.
- The general format of a class template is shown below:

```
template <class T>
class class_name
{
        //class member declaration with T type
        //...
};
```

- The class template definition is very similar to an ordinary class definition except the prefix template <class T> and the use of type T. The prefix tells the compiler that we are going to declare a template and use T as a type name in declaration. Thus, Vector has become a parameterized class with the type T as its parameter. T may be substituted by any data type including the user-defined types.

Class Template cont.

- Now we can create vectors for holding different data types.

```
template<class T>
  class Vector
  {
   private:
    T* v;
    int size;
   public:
    Vector(int m)
    {
     size = m;
     v = new T[size];
     v[i] = 0;
    }
    T operator*(Vector &x)
    {
     T sum = 0;
     for(int I = 0;I < size; i++)
      sum += v[i] * x.v[i];
     return sum;
    }
  };

  Vector <int> v1(5); // int vector having 5 elements
  Vector <float> v2(10); // float vector having 10 elements
  Vector <complex> v3(2); //complex vector having 2 complex //numbers
```

Class Template cont.

- The syntax for defining an object of a template class is:

```
class_name <type> object_name(arg_list);
```

- The process of creating a specific class from a class template is called instantiation.

//program55, program56, program57

Class Template with Multiple Parameters

- We can use more than one generic data type in a class template as shown below:

```
template <class T1, class T2,…>
class class_name
{
        //…
};
```

//program58,

Standard Template Library (STL)

- In order to help the C++ users in generic programming, a set of general-purpose templatized classes (data structures) and functions (algorithms) that could be used as a standard approach for storing and processing of data.
- The collection of these generic data and functions is called the Standard Template Library (STL). Using STL can save considerable time and effort, and lead to high quality programs since we are reusing well-written and well-tested components defined in the STL.
- The STL components are defined in the namespace std. We must therefore use the using namespace directive as

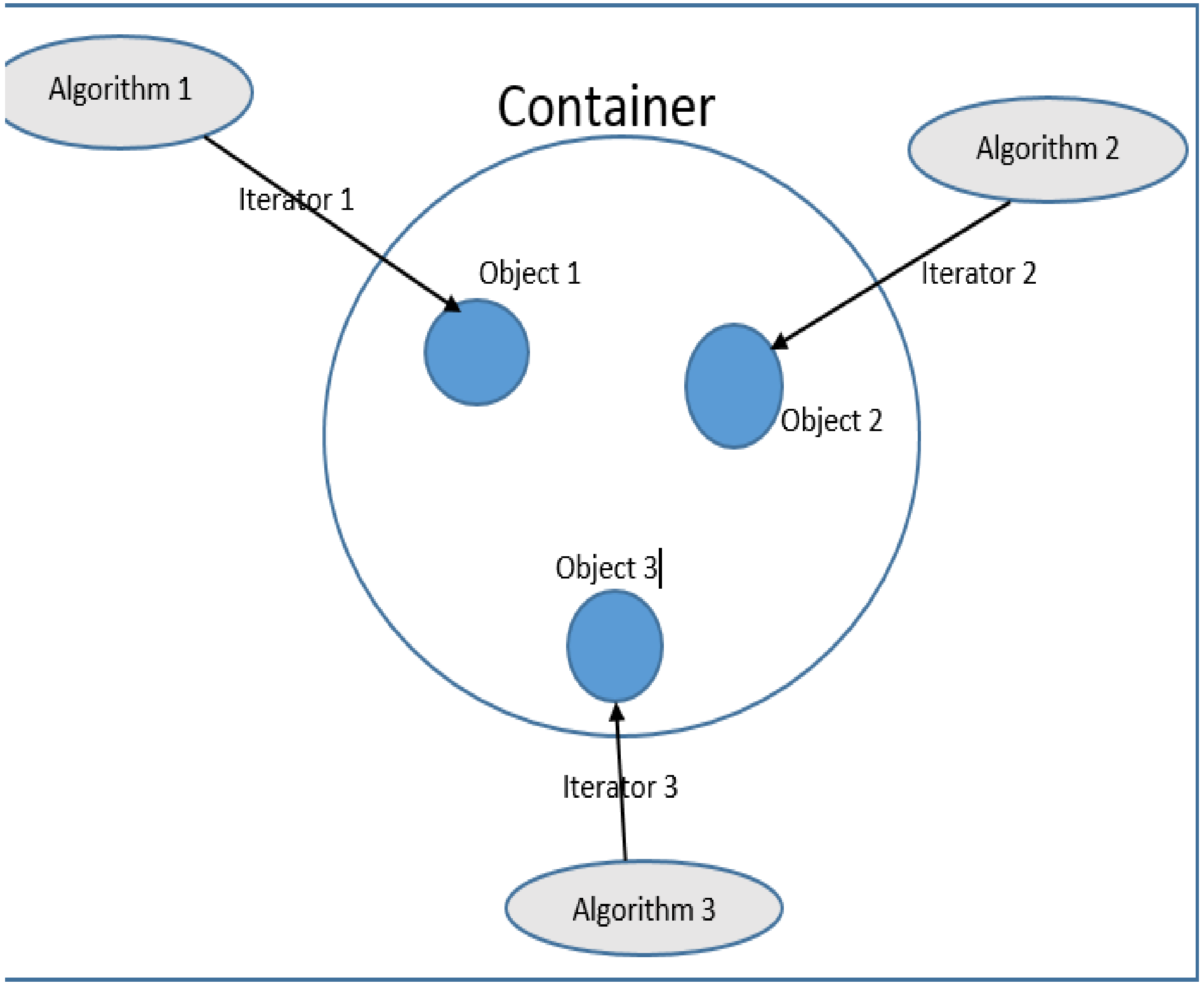
```
using namespace std;
```

to inform the compiler that we intend to use the Standard C++ Library.

- Basically, there are three key components of STL. They are:
  1. ***Containers***
  2. ***Algorithms***
  3. ***Iterators***

STL cont.

- A ***Container*** is an object that stores data. It is a way data is organized in memory. The STL containers are implemented by template classes and therefore can be easily customized to hold different types of data.
- An ***Algorithm*** is a procedure that is used to process the data contained in the containers. The STL includes many kinds of algorithms to provide support to tasks such as an initializing, searching, and merging. Algorithms are implemented by template functions.
- An ***Iterator*** is an object (like a pointer) that points to an element in a container. Iterators helps to move through the contents of container. Iterators are handled just like pointers which can be incremented and decremented them. Iterators connect algorithms with containers and play a key role in the manipulation of data stored in the containers.
- *Fig: Relationship between the three STL Components*



1. Container

- The STL defines ten containers which are grouped into three categories:

a. Sequence Containers
   - ➢ Vector
   - ➢ Deque
   - ➢ List

b. Associative Containers
   - ➢ Set
   - ➢ Multiset
   - ➢ Map
   - ➢ Multimap

c. Derived Containers
   - ➢ Stack
   - ➢ Queue
   - ➢ Priority Queue

- Each container class defines a set of functions that can be used to manipulate its contents. For example, a vector container defines functions for inserting elements, erasing the contents, and swapping the contents of two vectors.

2. Algorithm

- The algorithms are functions that can be used across a variety of containers for processing their contents. Although each container provides functions for its basic operations, STL provides more than sixty standard algorithms to support more extended or complex operations.
- STL algorithms are not member functions or friends of containers. They are standalone template functions.
- By using these algorithms programmer can save a lot of time and effort. To have access to STL algorithms, we must *include <algorithm>* in our program.

3. Iterator

- Iterators behave like pointers and are used to access container elements. They are often used to traverse from one element to another element.
- There are five types of iterators as

Iterator	Access Method	Direction of Movement	I/O Capability
Input	Linear	Forward only	Read only
Output	Linear	Forward only	Write only
Forward	Linear	Forward only	Read/Write
Bidirectional	Linear	Forward and Backward	Read/Write
Random	Random	Forward and backward	Read/Write

Vectors: An example of STL

- It stores elements in contiguous memory locations and enables direct access to any element using the subscript operator [ ]. A vector can change its size dynamically and therefore allocates memory as needed at runtime. The vector container supports random access iterators, and a wide range of iterator operations.

Function	**Tasks**
size()	Gives the number of elements
push_back()	Adds an element to the end
begin()	Gives the reference to the first element
insert()	Inserts elements in vector
erase()	Deletes specified elements

//program59

Exception Handling

- Two most common types of bugs are *Logic errors and Syntactic errors.* The logic errors occur due to poor understanding of the problem and solution procedure. The syntactic errors arise due to poor understanding of the language itself. We can detect these errors by using debugging and testing processes.
- We often come across some major problems other than we mentioned above which is known as *exceptions*. An exception is an unexpected condition (problem) that arise during the program execution (runtime).
- Moreover, exceptions are the runtime *anomalies or unusual conditions* that a program may encounter while executing. Anomalies might include conditions such as *division by zero, access to an array outside of its bounds, or running out of memory or disk space.*
- Exceptions are of two kinds, namely, *synchronous, and asynchronous exceptions*.
- Some examples of synchronous exceptions are index out of range, overflow, underflow, not being able to open a file, I/O errors etc.
- Some example of asynchronous exceptions is keyboard interrupt, hardware malfunction, disk failure etc. which are beyond the control of program. Exception handling mechanism provides a way to transfer control from one part of a program to another for synchronous exceptions only.
- In C++, exception handling is built on three keywords: *try, catch and throw.*

Exception Handling cont.

- `try:` A block of code which may cause exception is typically placed inside the try block. It is followed by one or more catch blocks. If an exception occurs, it is thrown by the try block.
- `catch:` This block catches the exception thrown from the try block. The command to handle the exception is written inside this catch block.
- `throw:` A program throws an exception when a program shows u
- Steps taken during exception handling
    1. *Hit the exception (detect the problem causing exception)*
    2. *Throw the exception*
    3. *Catch the exception*
    4. *Handle the exception*

Exception Handling cont.

- The general form of exception handling mechanism is shown below:

```
try  //block may have exception
{
     ...
     throw exception;//throwing exception
     ...
}
catch(type arg)//catches exceptions
{
     ...  //block of statements that handles the exception
     ...

}
```

//program60, program61

Multiple `catch` Statement

- It is possible that a program has more than one condition to throw an exception. In such case we can associate more than one catch statement with a try.

```
try{//try block}
catch(type1 arg){//catch block1}
catch(type2 arg){//catch block2}
...
catch(typeN arg){//catch blockN}
```

- When an exception is thrown, the exception handlers are searched for an appropriate match.
- The first handler that produce a match is executed. After executing the handler, the control goes to the first statement after the last catch block for that try.
- When no match is found, the program is terminated. It is possible that arguments of several catch statements match the type of an exception. In such cases, the first handler that matches the exception type is executed.

//program62