UNIT 2 TEMPLATES AND STANDARD TEMPLATE LIBRARY

Structure **Page Nos.**

2.0 INTRODUCTION

In the previous units, we have discussed about different features and functionalities of C++ programming language for object oriented programming. C++ being an object oriented language also supports reuse. Templates are one of the most prominent examples of reuse concept in action. This feature has been added to the C++ design recently. It supports the idea of generic programming by providing facility for defining generic classes and functions. Thus a template class provides a broad architecture which can be used to create a number of new classes. Similarly, a template function can be used to write various versions of the function. This chapter describes the template mechanism in C++ based on the paper titled 'Parameterized types for C++' by Bjarne Stroustup (creator of C++) published in Proceedings of the USENIX C++ Conference held in Denver, Colorado in October 1988.

This unit aims to explain the basic idea and motivation for templates. The unit explains in detail (with appropriate examples) the design and use of both Class and Function Templates. It then proceeds further to explain the use of templates and how they are related to the concepts of overloading and inheritance. We then discuss the structure and utility of the Standard Template Library in C++. Further, the unit describes the three core components of the Standard Template Library: containers, algorithms and iterators. These features are used in many real world application designs. The unit concludes with a summary of the Template and Standard template Library framework followed by model answers to assist you in check your progress exercises.

2.1 OBJECTIVES

At the end of the unit, you should be able to:

- identify the concept of Templates in C++;
- understand the definition and usage of class and function templates;
- write your own class and function templates;
- understand the design of class and function templates with and without parameters;
- visualize the idea behind design of the Standard Template Library;
- describe the components of the Standard Template Library and understand their use; and

- appreciate the use of the Standard template Library in real world application designs.

2.2 CLASS TEMPLATES

You might have got at least an idea from the introduction that templates are like stencils out of which we trace shapes. Function-template specializations and class-template specializations are like the separate tracings that all have the same shape, but could, for example, be drawn in different colours. In other words, a template may be considered as a kind of macro. When the actual object of that type is to be defined, the template definition is substituted with required data type. For example, if we define a template Array of elements, then this same generic definition may be used to create Array of integers or of characters or float quantities. We need not make a new class definition every time. We define a generic class with a parameter that s replaced by a particular data type at the time of actual use of that class. This is the reason template classes are also known as parameterized classes.

Designing a template class thus involves a simple process of creating a generic class with an anonymous type. The general syntax for defining a class template is:

```
template <class T>
class classname
{
……..
…….class specification with anonymous type T
……..
};
```

For example, consider the following template definition for a **Vector** class:

```
template<class T>
class vector
{
  T * v;  // the vector is of type T
  int size;
Public:
  vector(int m)
  {
    v = new T [size = m];
    for (int i=0; i<size; i++)
        v[i]=0;
   }
  vector (T * a)
  {
    for(int i=0; i<size; i++)
        v[i] = a[i];
   }
  T operator * (vector &x)
  {
   T sum = 0;
   for(int i=0; i<size; i++)
       sum += this->v[i] *x- v[i];
   return (sum);
   }
};
```

This definition creates a template class *vector* of type T with variables, constructors and '*' operator. This class definition is similar to an ordinary class definition except that of the use of **template<class T>** and use of type **T** inside the class definition. The template<class T> tells the compiler that it is a template class with parameter T instead of a normal class definition. The declaration thus creates a parameterized class with T as the parameter, which can be substituted with any valid data type. This can be done by a statement of the following form:

- classname <type> objectname(argument list);

For example, following statements create classes of 20 element integer and float vectors, respectively.

- vector <int> v(20);
- vector <float> v(20);

This task of creating an actual object from a template class is instantiation. Here we have written only one class definition for class *vector* but we have been able to create more than one actual instantiations of the template class *vector*.

A class template can also be created by using multiple generic data types as arguments. The general syntax for such definition would be as below:

```
template <class T1, class T2, …….>
class classname
{
……..
…….class specification with anonymous type T
……..
};
```

A simple program demonstrating the declaration and use of class templates is given below. Please note that this program instantiates two objects from the class template Example. The program first declares a template class Example with two arguments and then declares a constructor to instantiate the class. The main function creates two objects test1 and test2 of different types and their values are then displayed using the function show.

```
#include <iostream>

template<class T1, class T2>
class Example
{
  T1 x;
  T2 y;
 Public:
  Example(T1 a, T2 b)
  {
    x = a;
    y = b;
  }
  void show ()
  {
    cout << x << "and" << y << "\n";
  }
};
```

```
int main()
{
  Example <float, int> test1 (3.45, 345);
  Example <int, char> test2 (100, 'm');
  test1.show();
  test2.show();
return(0);
};
```

The program creates two template classes test1 and test2 using the template class Example. The test1 class has two parameter values "3.45" and "345", whereas test2 class has two parameter values "100" and character "m". For creating test1 object, arguments are float and integer respectively, whereas in case of test2 object they are integer and character. The values displayed in invocation of show() function from main will be "3.45 and 345" for test1 and "100 and m" for test2 object.

2.3 FUNCTION TEMPLATES

Function templates are similar to class templates in the sense that they create a generic function type. This generic function can then be used to create a family of functions that may take different arguments. A function template can be defined as follows:

```
template <class T>
return_type function_name (arguments of type T)
{
……..
…….body of function with argument of type T
……..
};
```

Function templates are another way of handling overloaded function requirements. If overloaded functions perform identical operations for different type of data then they can be more appropriately and conveniently declared as function templates. The following example demonstrates creation of a function template swap:

```
Template <class T>
void swap(T & x, T& y)
{
T temp = x;
x = y;
y = temp:
};
```

The swap() function can now be invoked like any ordinary function. Any call to swap() with input arguments will exchange the values contained in those arguments. Hence if a and b are integer variables and p and q are float variables; we may invoke swap() function as swap(a,b) and swap(p,q), respectively. The same function definition can be used to swap values of two variables of different types.

As we have discussed earlier, we can define class templates with multiple arguments, we can also define function templates with multiple arguments. This can be done through a declaration of the form:

```
template <class T1, class T2, ……..>
return_type function_name (arguments of type T1, T2,….)
{
……..
…….body of function
……..
};
```

The function and class templates can be used to write programs which work correctly on different types of data. For example, we can write the following program to sort a list in desired order using function templates swap() and bsort() as shown in the program below:

```
 #include <iostream>

template<class T>
void bsort(T a[], int n)
{
   for (int i=0; i<n-1; i++)
      for (int j=n-1; i<j; j--)
          if (a[j] < a[j-1])
             swap(a[j], a[j-1]);
 }
template <class X>
void swap( X &a, X &b)
{
   X temp =a;
   a = b;
   b = temp;
}


int main()
{
   int x[5] = {10,50,30,60,40};
   float y[5] = {3.2, 71.5, 17.3, 45.9, 92.7};

bsort(x,5);
bsort(y,5);

cout << "Sorted X-Array:";
for (int i=0; i<5; i++)
    cout << x[i] << " ";
cout << endl;

cout << "Sorted Y-Array:";
for (int j=0; j<5; j++)
    cout << y[j] << " ";
cout << endl;
return(0);
};
```

This program uses two function templates swap() and bsort(). The function template swap() is invoked within the bsort() function and is hence said to be nested in it. This program can be used to sort different types of lists without the need of modifying the program. The program will produce following output:
Sorted X-Array: 10 30 40 50 60

Sorted Y-Array: 3.2 17.3 45.9 71.5 92.7

A template function may also be overloaded in a manner similar to other functions. In fact, function templates and overloading are intimately related. All the function-template specializations generated from a function template have the same name, so the compiler uses overloading resolution to invoke the proper function. A function template can be overloaded in several ways:

a) functions having same name but different parameters
b) providing a non-template functions with the same function name but different arguments

Whenever, the compiler has to perform the matching process to determine what function to call, it achieves the overloading resolution as follows:

a) call an ordinary function that has an exact match
b) call a template function that could be created with an exact match
c) try normal overloading resolution to ordinary functions and call the one that matches.

In case no match is found, the compiler generates an error. In case there are multiple matches for the function call, the compiler considers the call to be ambiguous and the compiler generates an error message. It would also be worth mentioning that no automatic conversions are applied to arguments on the template functions.

☞ Check Your Progress 1

1) Fill in the blanks:

   a) Templates enable us to specify, with a single code segment, an entire range of related functions called ……..……, or an entire range of related classes called…….…… .

   b) All function template definitions begin with the keyword…… followed by a list of template parameters to the function template enclosed in…….. ..

   c) Class templates are also called ……..…… types.

   d) The …….. operator is used with a class template name to tie each member function definition to the class template's scope.

2) State whether following are *True* or *False*.

   a) A function template can be overloaded by another function template with the same function name.
   b) Template parameter names along template definitions must be unique.
   c) Each member function definition outside a class template must begin with a template header.
   d) Keywords *typename* and *class* as used with a template type parameter specifically mean "any user-defined class type."

3) Write appropriate statements to create a function template *printarray* that can display the values contained in array passed as parameter to the function. The function must be able to accept integer, float and character arrays as arguments.

   ……

   ……

   ……

4) Write appropriate statements to create a template class item that can instantiate objects of at least following types: item name: shirt, measure: size (expressed as characters 'S', 'M', 'L' and 'X') and item name: trouser, measure: size (expressed as waist size of integers). The template class must also have a template function to display the details of the items.

……

……

……

2.4 USE OF TEMPLATES

The concept of class templates and function templates derives its motivation from the principle of reuse. We have seen in earlier sections how templates allow us to create generic data types and functions that can fit into various situations. Rather than defining multiple classes and functions, we define a generic type and depending on the kind of input data it may customize itself. Templates in this sense serve as a blueprint for defining classes and functions. This not only eliminates code duplication for handling different data types but also makes the program development easier and more manageable. In the previous section, we saw an example of program for sorting that can a sort lists of various types. We present below another example demonstrating use of templates for solving a quadratic equation:

```
 #include <iostream>
#include <iomanip>
#include <cmath>

template<class T>
void roots(T a, T b, T c)
{
  T d = b*b - 4*a*c;
  if (d==0)
  {
    cout << "R1 = R2 ="  << -b/(2*a) << endl;
   }
  else if (d > 0)
   {
      cout << "Roots are real \n";
      float R = sqrt (d);
      float R1 = (-b + R) / (2*a);
      float R2 = (-b - R) / (2*a);
      cout << "R1 =" << R1 << "and" ;
      cout << "R2 =" << R2 << endl;
    }
   else
   {
      cout << "Roots are complex \n";
      float R1 = -b /(2*a);
      float R2 = sqrt (-d) / (2*a);
      cout << "Real part =" << R1 << endl;
      cout << "Imaginary part =" <<R2 <<endl;
    }
 }

int main()
```

```
{
  cout << "Integer coefficients \n";
  roots (1, -5, 6);
  cout << "\n Float coefficients \n";
  roots (1.5, 3.6, 5.0);

  return(0);
};
```

The program will generate following output:

```
Integer coefficients
Roots are real
R1 = 3 and R2 = 2
Float coefficients
Roots are complex
Real part = -1.2
Imaginary part = 1.375985
```

As we can see, the program above can be used to compute roots of a quadratic equation having different kinds of coefficients. It calculates roots for an equation having integer coefficient and for another equation having float coefficients. The templates have become so popular that now we have a rich library of predefined templates in C++, known as the Standard Template Library. We will discuss the contents and use of the standard template library in next section.

2.5 THE STANDARD TEMPLATE LIBRARY

The STL was developed by Alexander Stepanov and Meng Lee at Hewlett-Packard while pursuing their research in generic programming, with significant contributions from David Musser.

Recognizing that many data structures and algorithms are commonly used, the C++ standards committee added the Standard Template Library (STL) to the C++ standard library. The STL defines powerful, template-based, reusable components that implement many common data structures, and algorithms used to process those data structures. STL is a large collection of generic classes and functions. This large collection can be grouped at its core into three categories:

- Containers
- Algorithms, and
- Iterators.

These components work in conjunction with each other to provide solution to complex programming problems. A statement summarising their relationship could be "*Algorithms employ iterators to perform operations stored in containers*". The figure 2.1 further elaborates this relationship:

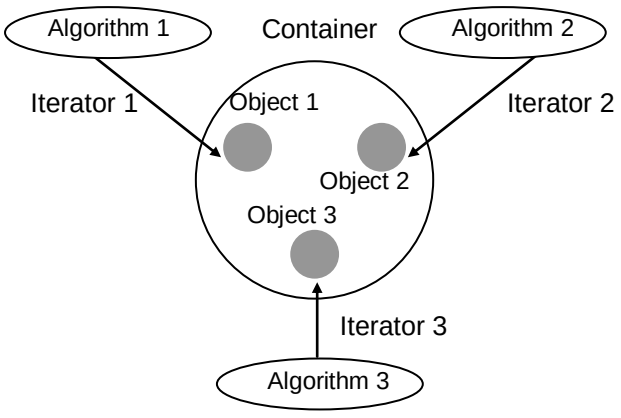

Figure 2.1 : Relationship between the three STL components

A *container* is an object that actually stores data. It is a way data is organized in memory. The STL containers are implemented by template classes and hence can be easily customized to hold different types of data. An *algorithm* is a procedure that is used to process the data stored in the containers. The STL include many different kinds of algorithms such as searching, sorting, copying, merging etc. Algorithms are implemented by template functions. An *iterator* is an object that works like a pointer. It is used to point to elements in a container. Iterator value may be incremented or decremented just like pointers. They play a key role in accessing and manipulating various data structures.

Containers

Containers are objects that hold data. These container classes are defined as class templates that can be customized to hold different kinds of data. The Figure 2.2 illustrates the three main types of container classes. The container classes contain definitions for commonly used data structures such as vector, list, queue, stack, set, map etc. Each container class also defines a set of functions that can be used to manipulate its contents. The STL defines a number of containers which can be grouped into mainly three types: sequence containers, associative containers and derived containers. The derived containers are sometime also referred to as container adapters.

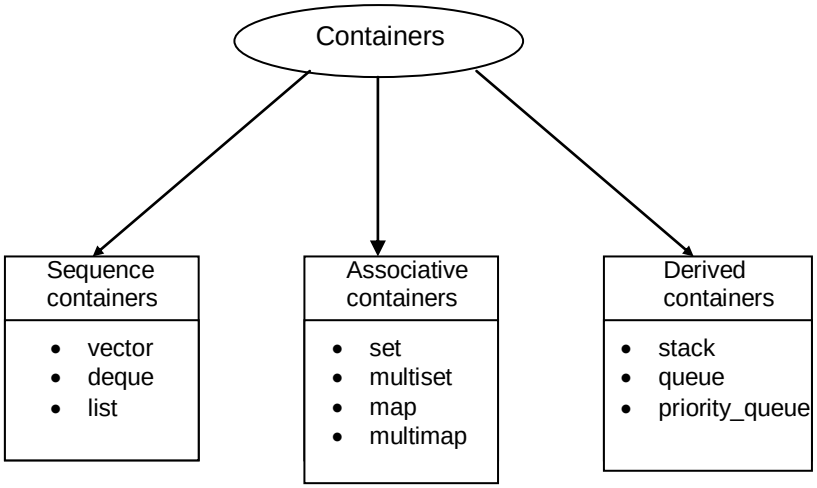

Figure 2.2 : Main Container types

The Table 2.1 lists some commonly used container classes available in the STL.

Table 2.1: List of commonly used Container

Container	Description	Header File	Iterator
vector	A dynamic array. Allows insertion and deletions at rear. Permits direct access.	<vector>	Random access
list	A bidirectional linear list. Allows insertion and deletions anywhere.	<list>	Bidirectional
stack	A standard stack, Last in First out operation.	<stack>	No iterator
queue	A standard Queue, First in First out operation.	<queue>	No iterator
priority queue	A priority queue, highest priority element as first out.	<queue>	No iterator
deque	A double ended queue, allows	<deque>	Random

	insertions and deletions at both ends.		access
set	An associative container for storing unique sets. Allows fast lookup.	<set>	Bidirectional
multiset	An associative container for storing non-unique sets.	<set>	Bidirectional
map	An associative container for storing unique key-value pairs.	<map>	Bidirectional
multimap	An associative container for storing key-value pairs that may use one-to-many mapping.	<map>	Bidirectional

The sequence containers represent linear data structures such as vectors and linked lists. The associative containers are non-linear container that typically store elements in a key-value pair fashion and support fast lookup. The sequence containers and associative containers are collectively referred to as *First Class containers*. Stacks and Queues are actually constrained versions of these first class containers and that is why often referred to as derived containers or container adapters. They enable a program to view a sequential container in a constrained manner. Sometimes we also hear about "near conatiners", which are similar to first class containers but do not support all functionalities of first class containers. Bitsets is one such example.

Hence, out of the various container classes listed in the table, vector, list and deque are sequential containers. Set, multiset, map and multimap are associative containers. Stack, queue and priority queue are derived containers.

Most STL containers provide similar functionality. Many generic operations, such as member function size, therefore apply to all containers. A good number of operations apply on subsets of container classes. Some of the common member functions that apply to most of container classes are listed in the Table 2.2:

Table 2.2: Some common member functions of Container Classes

Member Functions
default constructor, copy constructor, destructor, empty, insert, size, operator=, operator>, operator<, operator<=, operator>=, operator==, operator!=, swap
Functions found only in First class containers
max_size, begin, end, rbegin, rend, erase, clear.

Algorithms

Algorithms are functions that are used across a variety of container classes for processing their contents. As we have just learnt that each container class provides member functions for its basic operations, but STL further extends this by providing some standard algorithms for manipulating different containers. The STL contains approximately seventy standard algorithms to support more extended or complex operations. Standard algorithms have another advantage that they allow working with two different types of containers at the same time, unlike container member functions. The STL implement these algorithms as standalone function templates that can be customized to work with different kind of containers. Inserting, deleting, searching and sorting are some of the examples.

Unlike member functions, the algorithms operate on containers indirectly through the use of iterators. Many algorithms operate on sequences of elements defined by pair of iterators- one pointing to the first element of the sequence and other pointing to one

element past the last element. It is also possible to create new algorithms that operate in a similar fashion to that of STL algorithms. Algorithms often return iterators that indicate the results of algorithms (for example algorithm find). STL algorithms, based on the nature of operations they perform, may be categorized into following groups:

- Retrieve or non-modifying sequence algorithms
- Mutating-sequence algorithms
- Sorting Algorithms
- Set Algorithms
- Relational Algorithms

A list of some of these algorithms along with a description of their purpose is given in the Table 2.3:

Table 2.3: Mutating sequence Algorithm

Mutating-sequence algorithms	
copy()	Copies a sequence
copy_backward()	Copies a sequence from the end
fill()	Fills a sequence with a specified value
fill_n()	Fills first n elements with a specified value
generate()	Replaces all elements with the result of an operation
generate_n()	Replaces first n elements with the result of an operation
iter_swap()	Swaps elements pointed to by iterators
random_shuffle()	Places elements in random order
Remove()	Deletes elements of a specified value
remove_copy()	Copies a sequence after removing a specified value
remove_copy_if()	Copies a sequence after removing elements matching a predicate
remove_if()	Deletes elements matching matching a predicate
replace()	Replaces elements with a specified value
replace_copy()	Copies a sequence replacing elements with a given value
replace_copy_if()	Copies a sequence replacing elements matching a predicate

Non-modifying sequence algorithms	
adjacent_find()	Finds adjacent pair of objects that are equal
count()	Counts occurrence of a value in a sequence
count_if()	Counts number of elements that matches a predicate
equal()	True if two ranges are the same
find()	Finds first occurrence of a value in a sequence
find_end()	Finds last occurrence of a value in a sequence
find_first_of()	Finds a value from one sequence in another
find_if()	Finds first match of a predicate in a sequence
for_each()	Apply an operation to each element
mismatch()	Finds first elements for which two sequences differ
search()	Finds a subsequence within a sequence
search_n()	Finds a sequence of a specified number of similar elements

Sorting algorithms	
binary_search()	Conducts a binary search on an ordered sequence
equal_range()	Finds a sub range of elements with a given value
inplace_merge()	Merges two consecutive sorted sequences
lower_bound()	Finds the first occurrence of a specified value
make_heap()	Makes a heap from a sequence
merge()	Merges two sorted sequences
nth_element()	Puts a specified element in its proper place
partial_sort()	Sorts a part of a sequence
partial_sort_copy()	Sorts a part of a sequence and then copies
partition()	Places elements matching a predicate first
pop_heap()	Deletes the top element
push_heap()	Adds an element to heap
sort()	Sorts a sequence
sort_heap()	Sorts a heap
stable_partition()	Places elements matching a predicate first matching relative order
stable_sort()	Sorts maintaining order of equal elements
upper_bound()	Finds the last occurrence of a specified value

Set algorithms	
includes()	Finds whether a sequence is a subsequence of another
set_difference()	Constructs a sequence that is the difference of two ordered sets
set_intersection()	Constructs a sequence that contains the intersection of ordered sets
set_symmetric_difference()	Produces a set which is the symmetric difference between two ordered sets
set_union	Produces sorted union of two ordered sets

Relational algorithms	
equal()	Finds whether two sequences are the same
lexicographical_compare()	Compares alphabetically one sequence with other
max()	Gives minimum of two values
max_element()	Finds the maximum element within a sequence
min()	Gives minimum of two values
min_element()	Finds the minimum element within a sequence
Mismatch()	Finds the first mismatch between the elements in two sequence

Iterators

Iterators are used to access container class elements. They are called iterators because of their use in traversing the elements (from one to another) of a container class. In this sense they are quite similar to pointers. Iterators hold state information sensitive to the particular containers on which they operate, thus, iterators are implemented appropriately for each type of container. Certain iterator operations are uniform across containers. For example, ++ operation on an iterator moves it to the next element of the container. If iterator i points to a particular element, then, i++ points to the “next” element and *I refers to the element pointed by i.

There are five broad types of iterators supported by the STL. These are listed in Table 2.4:

Table 2.4: Types of Iterators

Iterator	Access method	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 & Backward	Read/ Write
Random	Random	Forward & Backward	Read/ Write

Each type of iterator is used for performing a particular set of functions. The input and output iterators are used to traverse a container and have functionality limited to this use. The forward operator also supports input and output and at the same time retaining its position in the container. The bidirectional iterator provides ability to move backwards in addition to forward movements. The random access iterator allows random jumps to a particular location in addition to bidirectional operations.

Examples of use of List and Map containers

Now that we had a look at the organization and different components of the STL, we will go through some illustrative examples of use of the STL components. Since the STL is quite big and we can not cover examples on the entire STL, we will see here one example each of the use of List and Map container classes.

List is a commonly used container class that implements a standard bidirectional linked list. It supports insertion and deletion operations and can be accessed only in a sequential manner. The STL class list provides appropriate set of member functions to manipulate lists. We will see here an example of use of list container class for creating and processing a list:

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

void display(list<int> &lst)
{
  list<int> :: iterator p;
  for (p=lst.begin(); p!=lst.end(); ++p)
        cout << *p << " " ;
  cout << "\n";
}

int main()
{
 list<int> list1; // empty list of zero length
 list<int> list2(5); //empty list of 5 elements

 for (int i=0; i<3; i++)
    list1.push_back(rand()/100);

list<int> :: iterator p;
for (p=list2.begin(); p!=list2.end(); ++p)
   *p=rand()/100;
cout << "list1 \n";
display(list1);
```

```
 cout << "list2 \n";
 display(list2);

//Add elements at both the ends of list1
  list1.push_front(100);
  list1.push_back(200);

 //Remove an element at front of list2
  list2.pop_front();

  cout << "now list1 \n";
  display(list1);
  cout << "now list2 \n";
  display(list2);

  list<int> listA, listB;
  listA=list1;
  listB=list2;

  //Merging two lists
  list1.merge(list2);
  cout << "Merged unsorted list \n";
  display(list1);

  //Sorting and Merging
  listA.sort();
  listB.sort();
  listA.merge(listB);
  cout << "Merged sorted list \n";
  display(listA);

  return(0);
};
```

The program above creates various lists and performs different operations on them. It uses member functions like begin(), end(), push_back(), push_front() etc. It also sorts and merges two lists. The user-defined function display() makes use of iterator to display the elements of the lists.

Another example that we would consider is that of container class map. As we discussed earlier a map is a sequence of key:value pairs, where a single value is associated with each unique key. Its an associative container class. The entries in a map are specified as:

phone ["ashok"] = 123456;

Here, phone is a map object and the statement associates number 123456 with the key value "ashok". The map entries can be manipulated using various member functions and algorithms. The key operations in a map include operations like add, delete, modify, sort the entries in map etc. We will have a look at an example of use of map in the program below:

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

```
typedef map<string, int> phonemap;

int main()
{
  string name;
  int number;
  phonemap phone;

 // Entering key:value pairs in map
  cout << "Enter three sets of name and numbers \n";
  for (int i=0, i<3; i++)
   {
      cin >> name;
      cin >> number;
      phone[name] = number;
    }

 // inserting a new entry
 phone["Ramesh"]= 621345;

  //inserting using insert() function
  phone.insert(pair<string, int> ("ajay", 234432));
  int n = phone.size();
  cout << "\n size of map:"  << n;

  // reading the entries in the map using iterator
  cout << "\n List of telephone numbers \n";
  phonemap::iterator p;
  for (p=phone.begin(); p!=phone.end(); p++)
       cout << (*p).first << "  " << (*p).second << "\n";

  cout << "\n";

  //looking up for an entry
  cout << "Enter name:";
  cin >> name;
  number = phone[name];
  cout << "Number:" << number << "\n";

  return(0);
};
```

This program first creates a map (phone) with three entries read from the keyboard. Then it moves to insert two new entries using two ways. It then prints the entire map using iterator. Finally it looks up the value of a given key stored in the map.

Similar to list and map container classes, the other container classes can also be used as the situation desires. We can use vector container class to create and manipulate various arrays; stack for handling LIFO memory operations; queue and priority queues for managing queue operations etc. We can manipulate these data structures not only by the member functions they contain, but also by using various algorithms that apply to them. Iterators help and guide the processing of elements contained in the class. The STL provides a rich set of template declarations along with different algorithms that are very useful in designing and implementing programming solutions for different real world problems. It supports and promotes reuse, a key theme of object oriented programming.

☞ Check Your Progress 2

1) Fill in the blanks:
   a) The two types of first-class STL containers are sequence containers and …….. containers.
   b) The five main iterator types are ……., ……..., ……., ……., and ……. .
   c) The three STL container adapters are ……., ……., and ……. .
   d) STL algorithms operate on container elements indirectly, using ……. .
   e) The sort algorithm requires a (n) …… iterator.

2) State whether following are *True* or *False*.
   a) The STL makes abundant use of inheritance and virtual functions.
   b) An iterator acts like a pointer to an element.
   c) STL algorithms can operate on C-like pointer-based arrays.
   d) STL algorithms are encapsulated as member functions within each container class.
   e) Container member function end yields the position of the container's last element.

3) For each of the following, write a single statement that performs the indicated task:
   a) Name the member functions that are used to refer to beginning and end of the list class.

      ……

      ……..……...……

      ……

   b) Name the different type names used to categorize the algorithms.

      ……

      ……

      ……

   c) What is the purpose of push_back(), push_front(), pop_back() and pop_front() functions of a list.

      ……

      ……

      ……

2.6 SUMMARY

This unit has introduced the basic idea of template classes and functions. Templates are mechanisms supported by C++ for generic programming. Templates allow us to generate a family of classes or a family of functions to handle different data types. Template classes and functions promote reuse and avoid code duplication. The member functions of a class template are also defined using the parameters of the class templates.

C++ now contains a rich set of template classes and functions packaged as a library, known as the standard template library. The STL consists of three main components: containers, algorithms, and iterators. Containers are objects that hold data of some type and are usually grouped into three types: sequential, associative and derived. Container classes contain a large number of member functions that make manipulating them simple. In addition to member functions, we also have a large number of algorithms (such as sorting, searching, copying, and merging) that are used to manipulate the container classes and perform various operations on them. Iterators, which are similar to pointers allow manipulation of elements of container classes indirectly by algorithms.

2.7 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

1. (a) function-template specialization, class-template specialization
   (b) template<…….>
   (c) parameterized
   (d) binary scope resolution

2. (a) True
   (b) False, it need not be unique.
   (c) True
   (d) False, This also allows for a type parameter of a fundamental type

3.
```
template<typename T>
        void printarray(T a[], int n)
        {
         for (int i=0; i<n-1; i++)
            cout << a[i] << " ";
         }
```

4.
```
template<class T1, class T2>
        class item
        {
          T1 x;
          T2 y;
        Public:
         item(T1 a, T2 b)
         {
           x = a;
           y = b;
         }
        template<typename T>
        void display(T a, T b)
        {
           cout << "item name" << a;
          cout << "item measure" << b ;
        }
```

Check Your Progress 2

1. (a) Associative
   (b) input, output, forward, bidirectional and random access
   (c) stack, queue, priority queue
   (d) iterators
   (e) random access

2. (a) False, These are avoided for performance reasons.
   (b) True
   (c) True
   (d) False, STL algorithms are not member functions. They operate indirectly on container classes through iterators.
   (e) False, it actually yields the position just after the end of the container.

3. (a) begin() and end()
   (b) non-modifying, mutating, sort, set and relational
   (c) push_bac() – is used to insert an element at the back of a list
   push_front() – is used to insert an element at the front of the list
   pop_back() – deleting an element from the back of the list
   pop_front() – deleting an element from the front of the list

2.8 FURTHER READINGS

1) E. Balaguruswamy, *Object Oriented Programming with C++*, Tata McGraw Hill, 2010.
2) P. Deitel and H. Deitel, C++: *How to Program, PHI*, 7th ed, 2010.
3) B. Strousstrup, *Programming – Principles and Practices using C++*, Addison Wesley, 2009.
4) B. Stroustrup, *"Parameterized types for C++" Proceedings of the USENIX C++* Conference, Denver, Colorado, October 1988.