GENERIC PROGRAMMING
STANDARD TEMPLATE LIBRARY

GENERIC PROGRAMMING

Introduction

To generate short, simple code and to avoid duplication of code, C++ provides *templates* to define the same piece of code for multiple data types. With templates, programmers can define a family of functions or classes that can perform operations on different types of data.

Templates comes under the category of meta-programming and auto code generation, where the generated code is not visible in general. Through templates, C++ supports generic programming.

Generic programming is a type of programming where the programmer specifies a general code first. That code is instantiated based on the type of parameters that are passed later in the program or at execution.

Entities such as a class or function created using generic programming are called generics.

Use of Templates

Following are the uses of templates in programming:

- Templates are widely used to implement the Standard Template Library (STL).
- Templates are used to create Abstract Data Types (ADTs) and classify algorithms and data structures.
- Class templates are generally used to implement containers.

Function Templates

A function template is a function which contains generic code to operate on different types of data. This enables a programmer to write functions without having to specify the exact type of parameters. Syntax for defining a template function is as follows:

```
template<class Type, ...>
return-type function-name(Type arg1, ...)
{
        //Body of function template
}
```

As shown above, the syntax starts with the keyword *template* followed by a list of template type arguments or also called generic arguments.

The *template* keyword tells the compiler that what follows is a template. Here, *class* is a keyword and *Type* is the name of generic argument.

Following program demonstrates a template function or function template:

```
#include <iostream>
using namespace std;
template<class Type>
void summ(Type x, Type y)
{
	cout<<"Sum is: "<<x+y<<endl;
}
int main()
{
	int a = 10, b = 20;
	summ(a, b);
	float p = 1.5, q = 2.4;
	summ(p, q);
	return 0;
}
```

Output for the above program is as follows:

```
Sum is: 30
Sum is: 3.9
```

In the above program compiler generates two copies of the above function template. One for int type arguments and the other for float type arguments. The function template can be invoked implicitly by writing summ(val1, val2) or explicitly by writing summ<int>(val1, val2).

Templates vs. Macros

As templates and macros perform similar tasks, we need know what is the difference between them. Following are the differences between templates and macros:

Templates	Macros
Code is simple and easier to understand	More chances of making mistakes
Easy to debug	Difficult to handle errors as they are handled by pre-processor
Being a function call, they are less efficient than macros	More efficient as they are compiled inline
As they is compile time checking, templates are considered type safe	Macros do not have any type checking and therefore are type unsafe

Guidelines for Using Template Functions

While using template functions, programmer must take care of the following:

Generic Data Types

Every template data type (or generic data type) should be used as types in the function template definition as type of parameters. If not used, it will result in an error. Consider the following example which leads to an error:

```
template<class Type>
void summ(int x, int y)          //Error since Type is not used for arguments x and y
{
        cout<<"Sum is: "<<x+y<<endl;
}
```

Also using some of the template data types is also wrong. Consider the following example which leads to an error:

```
template<class Type1, class Type2>
void summ(Type1 x, int y)     //Error since Type2 is not used
{
        cout<<"Sum is: "<<x+y<<endl;
}
```

Overloading Function Templates

As normal functions can be overloaded, template functions can also be overloaded. They will have the same name but different number of or type of parameters. Consider the following example which demonstrates template function overloading:

```
#include <iostream>
using namespace std;
void summ(int x, int y)
{
        cout<<"Normal Function: "<<endl;
        cout<<"Sum is: "<<x+y<<endl;
}
template<class Type1, class Type2>
void summ(Type1 x, Type2 y)
{
        cout<<"Template Function: "<<endl;
        cout<<"Sum is: "<<x+y<<endl;
}
int main()
{
        int a = 10, b = 20;
        summ(a, b);
        float p = 1.5, q = 2.4;
        summ(p, q);
        return 0;
}
```

Output for the above program is as follows:

```
Normal Function:
Sum is: 30
Template Function:
Sum is: 3.9
```

Whenever a compiler encounters the call to a overloaded function, first it checks if there is any normal function which matches and invokes it. Otherwise, it checks if there is any template function which matches and invokes it. If no function matches, error will be generated.

From the above example you can see that for the call *summ(a, b)*, normal function is invoked and for the call *summ(p, q)*, template function is invoked.

Recursive Template Functions

Like normal functions, template functions can also be called recursively. Consider the following example which demonstrates a recursive template function that calculates the factorial of a number:

```
#include <iostream>
using namespace std;
template<class Type>
Type fact(Type n)
{
	if(n==0 || n==1)
		return 1;
	else
		return n*fact(n-1);
}
int main()
{
	int n = 5;
	cout<<"Factorial of 5 is: "<<fact(5);
	return 0;
}
```

Output of the above program is as follows:

```
Factorial of 5 is: 120
```

Function Templates with User-defined Types

We can also pass user-defined types like class, structure, union as arguments to a function template. Consider the following function template which demonstrates passing a class as an argument:

```
#include <iostream>
using namespace std;
class Student
{
	public:
		int age;
		string name;
		Student(int age, string name)
		{
			this->age = age;
			this->name = name;
		}
};
template<class Type>
void display(Type &obj)
{
	cout<<"Name is: "<<obj.name<<endl;
	cout<<"Age is: "<<obj.age<<endl;
}
int main()
{
	Student s(25, "suresh");
	display(s);
	return 0;
}
```

Output for the above program is as follows:

```
Name is: suresh
Age is: 25
```

Class Template

Using templates programmers can create abstract classes that define the behavior of the class without actually knowing what data type will be handled by the class operations. Such classes are known as class templates. Syntax for creating a class template is as follows:

```
template<class Type1, class Type2, ...>
class ClassName
{
	...
	//Body of the class
	...
};
```

Syntax for creating an object of the template class is as follows:

```
ClassName<Type> ObjectName(params-list);
```

The process of creating an object of a template class or creating a specific class from a class template is called instantiation. The instantiated object of a template class is known as specialization. If a class template is specialized by some but not all parameters it is called partial specialization and if all the parameters are specialized, it is a full specialization.

Following program demonstrates creating a class template and using it:

```
#include <iostream>
using namespace std;
template<class Type>
class Swapper
{
	public:
		Type x, y;
	public:
		Swapper(Type x, Type y)
		{
			this->x = x;
			this->y = y;
		}
		void swap()
		{
			Type temp;
			temp = x;
			x = y;
			y = temp;
		}
		void display()
		{
			cout<<"x="<<x<<", y="<<y<<endl;
		}
};
int main()
{
	Swapper<int> iobj(10, 20);
	cout<<"Before swap:"<<endl;
	iobj.display();
	iobj.swap();
	cout<<"After swap:"<<endl;
	iobj.display();
	Swapper<float> fobj(10.3, 20.6);
	cout<<"Before swap:"<<endl;
	fobj.display();
	fobj.swap();
	cout<<"After swap:"<<endl;
	fobj.display();
	return 0;
}
```

Output for the above program is as follows:

```
Before swap:
x=10, y=20
After swap:
x=20, y=10
Before swap:
x=10.3, y=20.6
After swap:
x=20.6, y=10.3
```

The template data types allows default arguments. Syntax for specifying default data types is as follows:

```
template<class Type1, class Type2 = int>
class ClassName
{
        ...
        //Body of the class
        ...
};
```

We can define member functions of a class template outside the class. Syntax for doing it is as follows:

```
template<class Type>
return-type ClassName<Type> :: function-name(params-list)
{
        ...
        //Body of function
        ...
}
```

Class Template and Friend Functions

A friend function is a normal (non-member) function which can access the private members of a class. A function can be declared as a friend function inside the template class using the following syntax:

```
template<class Type>
class ClassName
{
        private:
                ...
        public:
                ...
                template<class Type1>
                friend return-type function-name(ClassName<Type1> Obj);
};
```

Following example demonstrates a friend function in class template:

```
#include <iostream>
using namespace std;
template<class Type>
class Swapper
{
	public:
		Type x, y;
	public:
		Swapper(Type x, Type y)
		{
			this->x = x;
			this->y = y;
		}
		template<class Type1>
		friend void swap(Swapper<Type1> &);
		void display()
		{
			cout<<"x="<<x<<", y="<<y<<endl;
		}
};
template<class Type1>
void swap(Swapper<Type1> &s)
{
	Type1 temp;
	temp = s.x;
	s.x = s.y;
	s.y = temp;
}
int main()
{
	Swapper<int> iobj(10, 20);
	cout<<"Before swap:"<<endl;
	iobj.display();
	swap(iobj);
	cout<<"After swap:"<<endl;
	iobj.display();
	Swapper<float> fobj(10.3, 20.6);
	cout<<"Before swap:"<<endl;
	fobj.display();
	swap(fobj);
	cout<<"After swap:"<<endl;
	fobj.display();
	return 0;
}
```

Output of the above program is as follows:

```
Before swap:
x=10, y=20
After swap:
x=20, y=10
Before swap:
x=10.3, y=20.6
After swap:
x=20.6, y=10.3
```

Class Templates and Static Variables

We know that static variables are shared by all the objects of a class. In class templates, all the objects of same type share the same static variable. Following program demonstrates a static variable inside a class template:

```
#include <iostream>
using namespace std;
template<class Type>
class Sample
{
	private:
		Type data;
	public:
		static int count;
		Sample()
		{
			count++;
		}
		static void show()
		{
			cout<<count<<endl;
		}
};
template<class Type>
int Sample<Type> :: count = 0;
int main()
{
	Sample<int> i1;
	Sample<int> i2;
	Sample<int> i3;
	Sample<char> c;
	Sample<double> d1;
	Sample<double> d2;
	cout<<"Number of integer objects: ";
	Sample<int>::show();
	cout<<"Number of char objects: ";
	Sample<char>::show();
	cout<<"Number of double objects: ";
	Sample<double>::show();
```

```
        return 0;
}
```

Output of the above program is as follows:

```
Number of integer objects: 3
Number of char objects: 1
Number of double objects: 2
```

Class Templates and Inheritance

A template class can be used in inheritance in the following ways:

- Deriving a class template from base class.
- Deriving a class template from a base class which also a class template and adding more template members to it.
- Deriving a class template from a base class which is a template class but disallowing the derived class and its derivatives to have template features.
- Deriving a non-template base class and adding some template members to it.

The syntax for declaring a derived class from base class which is a template class is as follows:

```
template<class Type1, class Type2>
class Base
{
        ...
        //Body of class
        ...
};
template<class Type1, class Type2>
class Derived : public Base<Type1, Type2>
{
        ...
        //Body of class
        ...
};
```

The derived class can be either a template class or a non-template class. If the derived class is a template class, then type of template arguments must be specified while creating its object. If the derived class is a non-template class, then type of template arguments must be specified during derivation itself as follows:

```
class Derived : public Base<int, float>
{
        ...
        //Body of class
        ...
};
```

Following example demonstrates creating a derived class which is a template class from a base class which also a template class:

```
#include <iostream>
using namespace std;
template<class Type1>
class Base
{
	protected:
		Type1 x;
	public:
		Base(Type1 x)
		{
			this->x = x;
		}
		void display()
		{
			cout<<"x = "<<x<<endl;
		}
};
template<class Type1, class Type2>
class Derived : public Base<Type1>
{
	private:
		Type2 y;
	public:
		Derived(Type1 x, Type2 y) : Base<Type1>(x)
		{
			this->y = y;
		}
		void display()
		{
			Base<Type1>::display();
			cout<<"y = "<<y<<endl;
		}
};
int main()
{
	Base<int> bobj(10);
	bobj.display();
	Derived<int,int> dobj(10, 20);
	dobj.display();
	return 0;
}
```

Output of the above program is as follows:

```
x = 10
x = 10
```

y = 20

Advantages and Disadvantages of Templates

The advantages of templates are as follows:

- Templates not only increases reusability of code but also makes the code short and simple.
- A single template can handle different types of parameters.
- Testing and debugging becomes simple.
- Templates support on-demand compilation i.e., only necessary template instances are compiled. In a program without templates all classes are compiled irrespective of the needed ones.

The disadvantages of templates are as follows:

- Some compilers doesn't support all template features. This reduces portability.
- Time needed to debug the code and resolve a error by the compiler needs more time.
- As templates are declared in the header, code is visible to all programs. This is in contradiction to information hiding.

Bubble Sort using Function Template

Following program performs Bubble sort using function templates:

```
#include <iostream>
using namespace std;
template<class Type1, class Type2>
void sort(Type1 arr[], Type2 n)
{
	for(int i = 0; i < n - 1; i++)
	{
		for(int j = 0; j < n - i - 1; j++)
		{
			if(arr[j] > arr[j+1])
			{
				Type1 temp = arr[j];
				arr[j] = arr[j+1];
				arr[j+1] = temp;
			}
		}
	}
}
int main()
{
```

```
	int n;
	cout<<"Enter the no. of elements: ";
	cin>>n;
	cout<<"Enter "<<n<<" integers: ";
	int iarr[n];
	for(int i=0; i<n; i++)
	{
		cin>>iarr[i];
	}
	cout<<"Enter "<<n<<" float values: ";
	float farr[n];
	for(int i=0; i<n; i++)
	{
		cin>>farr[i];
	}
	sort(iarr, n);
	sort(farr, n);
	cout<<"After sorting integer values are: ";
	for(int i=0; i<n; i++)
	{
		cout<<iarr[i]<<" ";
	}
	cout<<"\nAfter sorting floating point values are: ";
	for(int i=0; i<n; i++)
	{
		cout<<farr[i]<<" ";
	}
	return 0;
}
```

Input and Output for the above program are as follows:

```
Enter the no. of elements: 5
Enter 5 integers: 3 2 1 6 4
Enter 5 float values: 8.8 3.3 2.2 1.1 5.5
After sorting integer values are: 1 2 3 4 6
After sorting floating point values are: 1.1 2.2 3.3 5.5 8.8
```

Standard Template Library (STL)

Introduction

The Standard Template Library (STL) is a collection of components to implement data structures and frequently used operations on data. Three major components in STL are:

1. Containers
2. Algorithms
3. Iterators

Containers

Container is an object that can:

- Store data.
- Define how the data can be stored and manipulated using operations.

Containers are similar to data structures and are implemented using templates. So, a container can store data of different types. Examples of containers are: Vector, List, Dequeue, Set, MultiSet, Map, MultiMap, Stack, Queue, PriorityQueue etc.

Containers are of three types:

1. Sequence Containers
2. Associative Containers
3. Derived Containers

Sequence Containers

Data is stored in a linear fashion. Each element is related to other elements by its position. Elements can be accessed using an iterator. Examples of sequence containers are: Vector, List, and Dequeue.

Vector: Dynamic array that allows insertion and deletion of elements at any position. Elements in a vector are contiguous. Allows direct access of an element. Defined in header file <vector>.

List: A bi-directional list that allows insertion and deletion of elements at any position. Elements in a list are not contiguous. Defined in header file <list>.

Dequeue: A double ended queue which allows insertion and deletion at both ends. Allows direct access of any element. Defined in header file <deque>.

Associative Containers

There is no sequential ordering of elements. Data is sorted while giving input. Supports direct access of elements using keys. Data is stored as a tree. They facilitate fast searching, insertion, and deletion. Not efficient for sorting and random access. Examples of associative containers are: Set, multiset, Map, and multimap.

Set and multiset: Store multiple elements and provide functions to manipulate them using their keys. A set does not allow duplicate elements. But a multiset allows duplicate elements. They are defined in the header file <set>.

Map and multimap: Stores the elements as pair of key and value. Key is used for indexing and sorting the values. Values can be manipulated using keys. Map does not allows multiple values for a key. But a multimap allows multiple values for a key. They are defined in the header file <map>. An example for multimap is dictionary.

Derived Containers

These containers are created from sequence containers. They don't support iterators. As there are no iterators, data cannot be manipulated. Examples of derived containers are: Stack, Queue, and PriorityQueue.

Stack: Data is stored in Last In First Out (LIFO) order. Insertion and deletion of elements can be done only at one end.

Queue: Data is stored in First In First Out (FIFO) order. Insertion is done at one end and deletion is done at the other end.

PriorityQueue: The first element to be taken out is the element with highest priority.

Algorithms

The STL algorithms contains several functions that can be reused to perform several operations on the containers. Although each container provided basic functions to manipulate the data, STL provides around 60 algorithms (functions) which provided extended or complex functionality. These algorithms are categorized into five types as follows:

Retrieve or non-mutating algorithms

Function	Purpose
for_each	Applies a piece of code for each element in the range
count / count_if	Gives the number of elements that satisfies a specific criteria
equal	Determines if two sets of element are same
find / find_if / find_if_not	Finds the first element that meets the criteria
find_end	Finds the last sequence of elements in a certain range
find_first_of	Searches for any one of a set of elements
adjacent_find	Finds the first two adjacent items that are equal

search	Searches in a range of elements
search_n	Searches for a sequence in a range of elements

Mutating Algorithms

Function	Purpose
copy / copy_if / copy_n	Copies elements to a new location
copy_backward	Copies elements in reverse order
fill	Assigns elements a certain value
fill_n	Assigns a value to a specified number of elements
remove / remove_if	Removes elements
remove_copy / remove_copy_if	Copies elements after removing certain elements
replace / replace_if	Replaces elements that matches the condition
replace_copy / replace_copy_if	Copies elements after replacing with specified value
swap	Swaps the values of two elements
swap_ranges	Swaps two ranges of elements
reverse	Reverses the order of elements
reverse_copy	Copies elements in reverse order
rotate	Rotates the order of elements
rotate_copy	Copies and rotate elements
random_shuffle / shuffle	Randomly reorder elements
unique	Removes consecutive duplicate elements
unique_copy	Copies after removing duplicate elements

Sorting Algorithms

Function	Purpose
partition	Divides a range of elements into two groups
is_sorted	Checks whether a range is sorted into ascending order
is_sorted_until	Finds the largest sorted subrange
sort	Sorts a range into ascending order
partial_sort	Sorts the first N elements of a range
partial_sort_copy	Copies and partially sorts elements in a range
stable_sort	Sorts a range of elements while preserving order between equal elements
lower_bound	Finds the first occurrence of a value
upper_bound	Finds the last occurrence of a value
merge	Merges two sorted sequences of elements

Set Algorithms

Function	Purpose
includes	Returns true if one set is a subset of another
set_difference	Finds the difference between two sets
set_intersection	Finds the intersection between two sets
set_symmetric_difference	Finds the symmetric difference between two sets

set_union	Finds the union of two sets

Relational Algorithms

Function	Purpose
max	Returns the larger of two elements
max_element	Returns the largest element in a range
min	Returns the smaller of two elements
min_element	Returns the smallest elements in a range
lexicographical_compare	Returns true if one range is lexicographically less than another
adjacent_difference	Finds the differences between adjacent elements in a range
partial_sum	Finds the partial sum of a range of elements

Iterators

An iterator is an entity that allows programs to traverse the data in a container. Iterators are generally used to traverse the elements in a container. All containers except derived containers provides two types of containers:

container :: iterator (which provides a read/write iterator)
container :: const_iterator (which provides a read-only iterator)

STL provides five types of iterators as shown below for all containers except for derived containers.

Type	Direction	Access Type	Function	Ability to be saved
Input	Forward	Linear	Read	No
Output	Forward	Linear	Write	No
Forward	Forward	Linear	Read and Write	Yes
Bidirectional	Forward backward	Linear	Read and Write	Yes
Random	Forward backward	Random	Read and Write	Yes

All container classes provides four basic functions which can be used with the assignment operator. These functions are as follows:

begin() - Returns an iterator to the beginning of elements in the container.
cbegin() - Returns a read-only iterator to the beginning of elements in the container.
end() - Returns an iterator just past the end of the elements.
cend() - Returns a read-only iterator just past the end of the elements.

Vector

A vector is like a dynamic array where all the elements are stored contiguously. Elements can be added or removed at run-time as and when needed. Vector provides the following functions for working with the data stored in it:

Function	Purpose
push_back	Adds an element at the end
erase	Deletes elements
insert	Inserts elements
size	Gives the number of elements
at	Gives a reference to the specified element
back	Gives a reference to the last element
begin	Gives a reference to the first element
capacity	Gives the capacity of a vector
clear	Deletes all elements from the vector
empty	Checks if the vector is empty
end	Gives a reference to the end of the vector
pop_back	Deletes the last element
resize	Changes the size of the vector
swap	Interchanges the values stored at specified locations

Following program demonstrates working with a vector:

```
#include<iostream>
#include<vector>
using namespace std;
//print function to print the elements in the vector
void print(vector<int> &v)
{
	cout<<"Elements in the vector are: ";
	for(int i=0; i<v.size(); i++)
	{
		cout<<v[i]<<" ";
	}
	cout<<endl;
}
int main()
{
	vector<int> myvector;
	int num;
	cout<<"Enter 5 elements to store into vector: ";
	for(int i=0; i<5; i++)
	{
		cin>>num;
		myvector.push_back(num);  //inserts elements at the end
	}
	print(myvector);
	//Print the size (no. of elements) of vector
	cout<<"Size of vector is: "<<myvector.size()<<endl;
	//Print the capacity of vector
	cout<<"Capacity of vector is: "<<myvector.capacity()<<endl;
	//Get an iterator which points to the first element
	vector<int> :: iterator itr = myvector.begin();
	//Insert an element at index 2
```

```
        myvector.insert(itr+2, 8);
        cout<<"After inserting: "<<endl;
        print(myvector);
        //Delete an element at index 4
        myvector.erase(itr+4);
        cout<<"After deletion: "<<endl;
        print(myvector);
}
```

Input and output for the above program is as follows:

```
Enter 5 elements to store into vector: 1 4 6 9 2
Elements in the vector are: 1 4 6 9 2
Size of vector is: 5
Capacity of vector is: 8
After inserting:
Elements in the vector are: 1 4 8 6 9 2
After deletion:
Elements in the vector are: 1 4 8 6 2
```

List

A list (linked list) is a container in which an element points to next element and previous element. Direct access is not possible in a list. To reach nth element, we have to traverse all n-1 elements. The list container provides the following functions:

Function	**Purpose**
empty	Tests whether list is empty or not
size	Returns size of the list
max_size	Returns maximum size of the list
front	Access the first element
back	Access the last element
push_front	Inserts the element at the beginning
pop_front	Deletes the first element
push_back	Inserts the element at the end
insert	Inserts elements
erase	Deletes elements
swap	Swaps elements
resize	Changes the size of the list
clear	Removes all the elements
remove	Removes a specific element
remove_if	Removes elements that match a specific condition
merge	Merges two lists
unique	Removes duplicate elements
sort	Sorts the list
reverse	Reverses the list

Following program demonstrates working with a list:

```
#include<iostream>
#include<list>
using namespace std;
//print function to print the elements in a list
void print(list<int> &tmplist)
{
	list<int> :: iterator itr;
	for(itr=tmplist.begin(); itr!=tmplist.end(); itr++)
	{
		cout<<*itr<<" ";
	}
	cout<<endl;
}
int main()
{
	list<int> list1;
	list<int> list2;
	int num;
	cout<<"Enter 5 numbers into list 1: ";
	for(int i=0; i<5; i++)
	{
		cin>>num;
		list1.push_back(num);//inserts elements into list
	}
	cout<<"Enter 5 numbers into list 2: ";
	for(int i=0; i<5; i++)
	{
		cin>>num;
		list2.push_back(num);//inserts elements into list
	}
	cout<<"Elements in list 1 are: ";
	print(list1);
	cout<<"Elements in list 2 are: ";
	print(list2);
	//Sorting list 1
	list1.sort();
	cout<<"After sorting, elements in list 1 are: ";
	print(list1);
	//Reversing list 2
	list2.reverse();
	cout<<"After reversing, elements in list 2 are: ";
	print(list2);
	//Merging two lists
	list1.merge(list2);
	cout<<"After merging list2 with list1, elements in list 1 are: ";
	print(list1);
	return 0;
}
```

Input and output for the above program is as follows:

```
Enter 5 numbers into list 1: 5 4 3 2 1
Enter 5 numbers into list 2: 6 7 8 9 10
Elements in list 1 are: 5 4 3 2 1
Elements in list 2 are: 6 7 8 9 10
After sorting, elements in list 1 are: 1 2 3 4 5
After reversing, elements in list 2 are: 10 9 8 7 6
After merging list2 with list1, elements in list 1 are: 1 2 3 4 5 10 9 8 7 6
```

Maps

A map is like an associative array where each element is made of a pair. A pair contains a key and a value. Key serves as an index. The entries in a map are automatically sorted based on key when data is entered. Map container provides the following functions:

Function	Purpose
begin	Gives a reference to the first element
end	Gives a reference to the end of map
clear	Clears the map
empty	Checks whether the map is empty or not
erase	Delete elements in the map
find	Find whether a given element is in the map or not
insert	Inserts elements into the map
Size	Returns the size of map

Following program demonstrates working with a map:

```
#include<iostream>
#include<map>
using namespace std;
int main()
{
	map<string,int> STDcode;
	string area;
	int code;
	for(int i=1; i<=5; i++)
	{
		cout<<"Enter city: ";
		getline(cin, area);
		cout<<"Enter STD code: ";
		cin>>code;
		fflush(stdin);
		STDcode[area] = code;
	}
	STDcode["Chennai"] = 56;    //First way to insert
	STDcode.insert(pair<string,int>("Banglore",57));    //Second way to insert
```

```
        map<string,int> :: iterator itr;
        cout<<"Map entries are: "<<endl;
        //Printing map entries
        for(itr = STDcode.begin(); itr!=STDcode.end(); itr++)
        {
                cout<<(*itr).first<<", "<<(*itr).second<<endl;
        }
        return 0;
}
```

Input and output for the above program is as follows:

```
Enter city: Hyderabad
Enter STD code: 51
Enter city: Vizag
Enter STD code: 52
Enter city: Guntur
Enter STD code: 53
Enter city: Simla
Enter STD code: 54
Enter city: Goa
Enter STD code: 55
Map entries are:
Banglore, 57
Chennai, 56
Goa, 55
Guntur, 53
Hyderabad, 51
Simla, 54
Vizag, 52
```