Chapter 9

Standard Template Library

The **Standard Template Library** (**STL**) is a C++ software library that influenced many parts of the C++ Standard Library It provides four components called algorithms, containers, functional and iterators. The STL provides a ready-made set of common classes for C++, such as containers and associative arrays, that can be used with any built-in type and with any user-defined type that supports some elementary operations (such as copying and assignment). STL algorithms are independent of containers, which significantly reduces the complexity of the library.

The STL achieves its results through the use of templates. This approach provides compile-time polymorphism that is often more efficient than traditional run-time polymorphism. Modern C++ compilers are tuned to minimize any abstraction penalty arising from heavy use of the STL.

At the core of the standard template library are three foundational items: containers, algorithms, and iterators. These items work in conjunction with one another to provide off-the-shelf solutions to a variety of programming problems.

Containers

Containers are objects that hold other objects, and there are several different types. For example, the **vector** class defines a dynamic array, **deque** creates a double-ended queue, and **list** provides a linear list. These containers are called sequence containers because in STL terminology, a sequence is a linear list. In addition to the basic containers, the STL also defines associative containers, which allow efficient retrieval of values based on keys. For example, a **map** provides access to values with unique keys. Thus, a **map** stores a key/value pair and allows a value to be retrieved given its key. Each container class defines a set of functions that may be applied to the container. For example, a list container includes functions that insert, delete, and merge elements. A stack includes functions that push and pop values.

Algorithms

Algorithms act on containers. They provide the means by which you will manipulate the contents of containers. Their capabilities include initialization, sorting, searching, and transforming the contents of containers. Many algorithms operate on a range of elements within a container.

Iterators

Iterators are objects that are, more or less, pointers. They give you the ability to cycle through the contents of a container in much the same way that you would use a pointer to cycle through an array. There are five types of iterators:

Iterator	Access Allowed
Random	Access Store and retrieve values. Elements may be accessed randomly.
Bidirectional	Store and retrieve values. Forward and backward moving.
Forward	Store and retrieve values. Forward moving only.

Input	Retrieve, but not store values. Forward moving only.
Output	Store, but not retrieve values. Forward moving only.

In general, an iterator that has greater access capabilities can be used in place of one that has lesser capabilities. For example, a forward iterator can be used in place of an input iterator. Iterators are handled just like pointers. You can increment and decrement them.
You can apply the * operator to them. Iterators are declared using the **iterator** type defined by the various containers.The STL also supports reverse iterators. Reverse iterators are either bidirectional or random-access iterators that move through a sequence in the reverse direction. Thus, if a reverse iterator points to the end of a sequence, incrementing that iterator will cause it to point to one element before the end.

Other STL Elements

In addition to containers, algorithms, and iterators, the STL relies upon several other standard components for support. Chief among these are allocators, predicates, comparison functions, and function objects. Each container has defined for it an allocator. Allocators manage memory allocation for a container. The default allocator is an object of class **allocator**, but you can define your own allocators if needed by specialized applications. For most uses, the default allocator is sufficient. Several of the algorithms and containers use a special type of function called a predicate. There are two variations of predicates: unary and binary. A unary predicate takes one argument, while a binary predicate has two.

Vectors

Perhaps the most general-purpose of the containers is **vector**. The **vector** class supports a dynamic array. This is an array that can grow as needed. As you know, in C++ the size of an array is fixed at compile time. While this is by far the most efficient way to implement arrays, it is also the most restrictive because the size of the array cannot be adjusted at run time to accommodate changing program conditions. A vector solves this problem by allocating memory as needed. Although a vector is dynamic, you can still use the standard array subscript notation to access its elements.
The template specification for **vector** is shown here:

```
template <class T, class Allocator = allocator<T>> class vector
```

Here, **T** is the type of data being stored and **Allocator** specifies the allocator, which defaults to the standard allocator. **vector** has the following constructors:

```
explicit vector(const Allocator &a = Allocator( ) );
explicit vector(size_type num, const T &val = T ( ),
const Allocator &a = Allocator( ));
vector(const vector<T, Allocator> &ob);
template <class InIter> vector(InIter start, InIter end,
const Allocator &a = Allocator( ));
```

The first form constructs an empty vector. The second form constructs a vector that has num elements with the value val. The value of val may be allowed to default. The third form constructs a vector that contains the same elements as ob. The fourth form constructs a vector that contains the elements in the range specified by the iterators start and end.

Any object that will be stored in a **vector** must define a default constructor. It must also define the < and == operations. Some compilers may require that other comparison operators be defined. (Since implementations vary, consult your compiler's documentation for precise information.) All of the built-in types automatically satisfy these requirements.

Although the template syntax looks rather complex, there is nothing difficult about declaring a vector. Here are some examples:

```
vector<int> iv; // create zero-length int vector
vector<char> cv(5); // create 5-element char vector
vector<char> cv(5, 'x'); // initialize a 5-element char vector
vector<int> iv2(iv); // create int vector from an int vector
```

The following comparison operators are defined for **vector**:

==, <, <=, !=, >, >=

The subscripting operator [ ] is also defined for **vector**. This allows you to access the elements of a vector using standard array subscripting notation.

Program 9.1 **Basic operation of a vector.**

Solution:

```
#include <iostream>
#include <vector>
#include <cctype>
using namespace std;
void main ()
{
        vector<char> v(10); // create a vector of length 10
        int i;
        // display original size of v
        cout << "Size = " << v.size() << endl;
        // assign the elements of the vector some values
        for(i=0; i<10; i++) v[i] = i + 'a';
        // display contents of vector
        cout << "Current Contents:\n";
        for(i=0; i<v.size(); i++) cout << v[i] << " ";
        cout << "\n\n";
        cout << "Expanding vector\n";
        /* put more values onto the end of the vector,
        it will grow as needed */
        for(i=0; i<10; i++) v.push_back(i + 10 + 'a');
```

```
        // display current size of v
        cout << "Size now = " << v.size() << endl;
        // display contents of vector
        cout << "Current contents:\n";
        for(i=0; i<v.size(); i++) cout << v[i] << " ";
        cout << "\n\n";
        // change contents of vector
        for(i=0; i<v.size(); i++) v[i] = toupper(v[i]);
        cout << "Modified Contents:\n";
        for(i=0; i<v.size(); i++) cout << v[i] << " ";
        cout << endl;
}
```

Output:

Size = 10
Current Contents:
a b c d e f g h i j
Expanding vector
Size now = 20
Current contents:
a b c d e f g h i j k l m n o p q r s t
Modified Contents:
A B C D E F G H I J K L M N O P Q R S T

Assignment 9

Short Type Questions

1. Differentiate between new and malloc.
2. Differentiate between delete and free.

Long Type Questions

3. Define dynamic memory management. Explain dynamic memory management in C++ with suitable example.