14

Introduction to the Standard Template Library

Key Concepts

- Software evolution
- Standard templates
- Standard C++ library
- Containers
- Sequence containers
- Associative containers
- Derived containers
- Algorithms
- Iterators
- Function object

14.1 Introduction

We have seen how templates can be used to create generic classes and functions that could extend support for generic programming. In order to help the C++ users in generic programming, Alexander Stepanov and Meng Lee of Hewlett-Packard developed 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 classes and functions is called the *Standard Template Library (STL)*. The STL has now become a part of the ANSI standard C++ class library.

STL is large and complex and it is difficult to discuss all of its features in this chapter. We therefore present here only the most important features that would enable the readers to begin using the STL effectively. Using STL can save considerable time and effort, and lead to high quality programs. All these benefits are possible because we are basically "reusing" the well-written and well-tested components defined in the STL.

multiset	An associate container for storing non-unique sets. (Duplicates allowed)	<set>	Bidirectional
map	An associate container for storing unique key/value pairs. Each key is associated with only one value (One-to-one mapping). Allows key-based lookup.	<map>	Bidirectional
multimap	An associate container for storing key/value pairs in which one key may be associated with more than one value (one-to-many mapping). Allows key-based lookup.	<map>	Bidirectional
stack	A standard stack. Last-in-first-out(LIFO).	<stack>	No iterator
queue	A standard queue. First-in-first-out(FIFO).	<queue>	No iterator
priority–queue	A priority queue. The first element out is always the highest priority element.	<queue>	No iterator

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.

Sequence Containers

Sequence containers store elements in a linear sequence, like a line as shown in Fig. 14.3. Each element is related to other elements by its position along the line. They all expand themselves to allow insertion of elements and all of them support a number of operations on them.

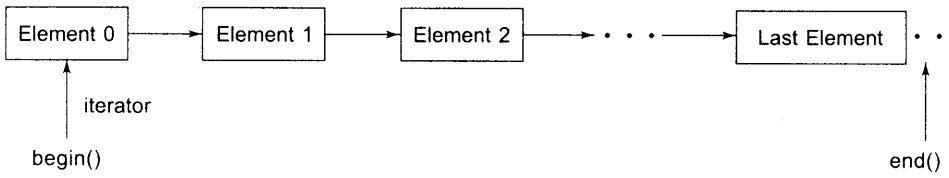

Fig. 14.3 ⇔ *Elements in a sequence container*

The STL provides three types of sequence containers:

- vector
- list
- deque

Elements in all these containers can be accessed using an iterator. The difference between the three of them is related to only their performance. Table 14.2 compares their performance in terms of speed of random access and insertion or deletion of elements.

Table 14.2 *Comparison of sequence containers*

Container	*Random access*	*Insertion or deletion in the middle*	*Insertion or deletion at the ends*
vector	Fast	Slow	Fast at back
list	Slow	Fast	Fast at front
deque	Fast	Slow	Fast at both the ends

Associative Containers

Associative containers are designed to support direct access to elements using keys. They are not sequential. There are four types of associative containers:

- set
- multiset
- map
- multimap

All these containers store data in a structure called *tree* which facilitates fast searching, deletion, and insertion. However, these are very slow for random access and inefficient for sorting.

Containers **set** and **multiset** can store a number of items and provide operations for manipulating them using the values as the *keys*. For example, a **set** might store objects of the **student** class which are ordered alphabetically using names as keys. We can search for a desired student using his name as the key. The main difference between a **set** and a **multiset** is that a **multiset** allows duplicate items while a **set** does not.

Containers **map** and **multimap** are used to store pairs of items, one called the *key* and the other called the *value*. We can manipulate the values using the keys associated with them. The values are sometimes called *mapped values*. The main difference between a **map** and a **multimap** is that a **map** allows only one key for a given value to be stored while **multimap** permits multiple keys.

Derived Containers

The STL provides three derived containers namely, **stack**, **queue**, and **priority_queue**. These are also known as *container adaptors*.

Stacks, queues and priority queues can be created from different sequence containers. The derived containers do not support iterators and therefore we cannot use them for data manipulation. However, they support two member functions **pop( )** and **push( )** for implementing deleting and inserting operations.

14.4 Algorithms

Algorithms are functions that can be used generally 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. Standard algorithms also permit us to work with two different types of containers at the same time. Remember, STL algorithms are not member functions or friends of containers. They are standalone template functions.

STL algorithms reinforce the philosophy of reusability. By using these algorithms, programmers can save a lot of time and effort. To have access to the STL algorithms, we must include **<algorithm>** in our program.

STL algorithms, based on the nature of operations they perform, may be categorized as under:

- Retrieve or non-mutating algorithms
- Mutating algorithms
- Sorting algorithms
- Set algorithms
- Relational algorithms

These algorithms are summarized in Tables 14.3 to 14.7. STL also contains a few numeric algorithms under the header file **<numeric>**. They are listed in Table 14.8.

Table 14.3 *Non-mutating algorithms*

Operations	*Description*
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

Table 14.4 *Mutating algorithms*

Operations	*Description*
Copy( )	Copies a sequence
copy_backward( )	Copies a sequence from the end
fill( )	Fills a sequence with a specified value

(Contd)

Table 14.4 *Contd*

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 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
replace_if( )	Replaces elements matching a predicate
reverse( )	Reverses the order of elements
reverse_copy( )	Copies a sequence into reverse order
rotate( )	Rotates elements
rotate_copy( )	Copies a sequence into a rotated
swap( )	Swaps two elements
swap_ranges( )	Swaps two sequences
transform( )	Applies an operation to all elements
unique( )	Deletes equal adjacent elements
unique_copy( )	Copies after removing equal adjacent elements

Table 14.5 *Sorting algorithms*

Operations	***Description***
binary_search( )	Conducts a binary search on an ordered sequence
equal_range( )	Finds a subrange 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

Table 14.6 *Set algorithms*

Operations	*Description*
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

Table 14.7 *Relational algorithms*

Operations	*Description*
equal( )	Finds whether two sequences are the same
lexicographical_compare()	Compares alphabetically one sequence with other
max( )	Gives maximum 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 sequences

Table 14.8 *Numeric algorithms*

Operations	*Description*
accumulate( )	Accumulates the results of operation on a sequence
adjacent_difference( )	Produces a sequence from another sequence
inner_product( )	Accumulates the results of operation on a pair of sequences
partial_sum( )	Produces a sequence by operation on a pair of sequences

14.5 Iterators

Iterators behave like pointers and are used to access container elements. They are often used to traverse from one element to another, a process known as *iterating* through the container.

There are five types of iterators as described in Table 14.9.

Table 14.9 *Iterators and their characteristics*

Iterator	*Access method*	*Direction of movement*	*I/O capability*	*Remark*
Input	Linear	Forward only	Read only	Cannot be saved
Output	Linear	Forward only	Write only	Cannot be saved
Forward	Linear	Forward only	Read/Write	Can be saved
Bidirectional	Linear	Forward and backward	Read/Write	Can be saved
Random	Random	Forward and backward	Read/Write	Can be saved

Different types of iterators must be used with the different types of containers (See Table 14.1). Note that only sequence and associative containers are traversable with iterators.

Each type of iterator is used for performing certain functions. Figure 14.4 gives the functionality Venn diagram of the iterators. It illustrates the level of functionality provided by different categories of iterators.

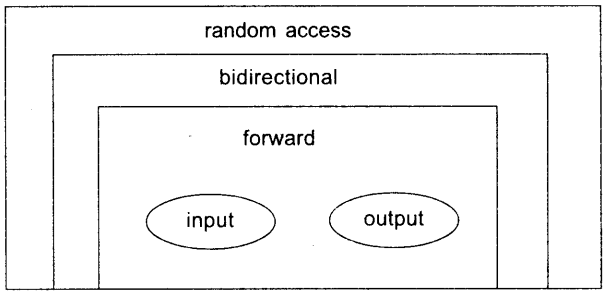

Fig. 14.4 ⇔ *Functionality Venn diagram of iterators*

The *input* and *output* iterators support the least functions. They can be used only to traverse in a container. The *forward* iterator supports all operations of input and output iterators and also retains its position in the container. A *bidirectional* iterator, while supporting all forward iterator operations, provides the ability to move in the backward direction in the container. A *random access* iterator combines the functionality of a bidirectional iterator with an ability to jump to an arbitrary location. Table 14.10 summarizes the operations that can be performed on each iterator type.

Table 14.10 *Operations supported by iterators*

Iterator	***Element access***	***Read***	***Write***	***Increment operation***	***Comparison***
Input	->	v = *p		++	==, !=
Output			*p = v	++	
Forward	->	v = *p	*p = v	++	==, !=
Bidirectional	->	v = *p	*p = v	++, - -	==, !=
Random access	->, [ ]	v = *p	*p = v	++, - -, +, -, +=, -=	==, !=, <, >, <=, >=

14.6 Application of Container Classes

It is beyond the scope of this book to examine all the containers supported in the STL and provide illustrations. Therefore, we illustrate here the use of the three most popular containers, namely, **vector**, **list**, and **map**.

```
    display(v);

    // Removing 4th and 5th elements
    v.erase(v.begin()+3,v.begin()+5);  // Removes 4th and 5th element

    // Display the contents
    cout << "\nContents after deletion: \n";
    display(v);
    cout << "END\n";
    return(0);
}
```

Program 14.1

Given below is the output of Program 14.1:

```
Initial size = 0

Enter five integer values:  1 2 3 4 5
Size after adding 5 values:  5
Current contents:
1  2  3  4  5

Size = 6
Contents now:
1  2  3  4  5  6

Contents after inserting:
1  2  3  9  4  5  6

Contents after deletion:
1  2  3  5  6
END
```

The program uses a number of functions to create and manipulate a vector. The member function **size()** gives the current size of the vector. After creating an **int** type empty vector **v** of zero size, the program puts five values into the vector using the member function **push_back()**. Note that **push_back()** takes a value as its argument and adds it to the back end of the vector. Since the vector **v** is of type **int**, it can accept only integer values and therefore the statement

```
v.push_back(6.6);
```

truncates the values 6.6 to 6 and then puts it into the vector at its back end.

The program uses an iterator to access the vector elements. The statement

```
vector<int> :: iterator itr = v.begin();
```

declares an iterator **itr** and makes it to point to the first position of the vector. The statements

```
itr = itr + 3;
v.insert(itr,9);
```

inserts the value 9 as the fourth element. Similarly, the statement

```
v.erase(v.begin()+3, v.begin()+5);
```

deletes 4^{th} and 5^{th} elements from the vector. Note that **erase(m,n)** deletes only n-m elements starting from m^{th} element and the n^{th} element is not deleted.

The elements of a vector may also be accessed using subscripts (as we do in arrays). Notice the use of **v[i]** in the function **display()** for displaying the contents of **v**. The call **v.size()** in the **for** loop of **display()** gives the current size of **v**.

Lists

The **list** is another container that is popularly used. It supports a bidirectional, linear list and provides an efficient implementation for deletion and insertion operations. Unlike a vector, which supports random access, a list can be accessed sequentially only.

Bidirectional iterators are used for accessing list elements. Any algorithm that requires input, output, forward, or bidirectional iterators can operate on a **list**. Class **list** provides many member functions for manipulating the elements of a list. Important member functions of the **list** class are given in Table 14.12. Use of some of these functions is illustrated in Program 14.2. Header file **<list>** must be included to use the container class **list**.

LISTING 14.2

```
#include <iostream>
#include <list>
#include <cstdlib>    // For using rand() function

using namespace std;

void display(list<int> &lst)
{
        list<int> :: iterator p;
```

(Contd)

```
        for(p = lst.begin(); p != lst.end(); ++p)
                cout << *p << ", ";
        cout << "\n\n";
}

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

        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 two elements at the ends of list1
        list1.push_front(100);
        list1.push_back(200);

        // Remove an element at the 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(unsorted)
        list1.merge(list2);
        cout << "Merged unsorted lists \n";
        display(list1);
```

(Contd)

```
    // Sorting and merging
    listA.sort();
    listB.sort();
    listA.merge(listB);
    cout << "Merged sorted lists \n";
    display(listA);

    // Reversing a list
    listA.reverse();
    cout << "Reversed merged list \n";
    display(listA);

    return(0);
}
```

Program 14.2

Output of the Program 14.2 would be:

```
List1
0, 184, 63,

List2
265, 191, 157, 114, 293,

Now List1
100, 0, 184, 63, 200,

Now List2
191, 157, 114, 293,

Merged unsorted lists
100, 0, 184, 63, 191, 157, 114, 200, 293,

Merged sorted lists
0, 63, 100, 114, 157, 184, 191, 200, 293,

Reversed merged list
293, 200, 191, 184, 157, 114, 100, 63, 0,
```

The program declares two empty lists, **list1** with zero length and **list2** of size 5. The **list1** is filled with three values using the member function **push_back()** and math function rand(). The **list2** is filled using a **list** type iterator **p** and a **for** loop. Remember that

list2.begin() gives the position of the first element while **list2.end()** gives the position immediately after the last element. Values are inserted at both the ends using **push_front()** and **push_back()** functions. The function **pop_front()** removes the first element in the list. Similarly, we may use **pop_back()** to remove the last element.

The objects of list can be initialized with other list objects like

```
listA = list1;
listB = list2;
```

The statement

```
list1.merge(list2);
```

simply adds the **list2** elements to the end of **list1**. The elements in a list may be sorted in increasing order using **sort()** member function. Note that when two sorted lists are merged, the elements are inserted in appropriate locations and therefore the merged list is also a sorted one.

We use a **display()** function to display the contents of various lists. Note the difference between the implementations of **display()** in Program 14.1 and Program 14.2.

Table 14.12 *Important member functions of the list class*

Function	*Task*
back( )	Gives reference to the last element
begin( )	Gives reference to the first element
clear( )	Deletes all the elements
empty( )	Decides if the list is empty or not
end( )	Gives reference to the end of the list
erase( )	Deletes elements as specified
insert( )	Inserts elements as specified
merge( )	Merges two ordered lists
pop_back( )	Deletes the last element
pop_front( )	Deletes the first element
push_back( )	Adds an element to the end
push_front( )	Adds an element to the front
remove( )	Removes elements as specified
resize( )	Modifies the size of the list
reverse( )	Reverses the list
size( )	Gives the size of the list
sort( )	Sorts the list
splice( )	Inserts a list into the invoking list
swap( )	Exchanges the elements of a list with those in the invoking list
unique( )	Deletes the duplicating elements in the list

Maps

A **map** is a sequence of (key, value) pairs where a single value is associated with each unique key as shown in Fig. 14.5. Retrieval of values is based on the key and is very fast. We should specify the key to obtain the associated value.

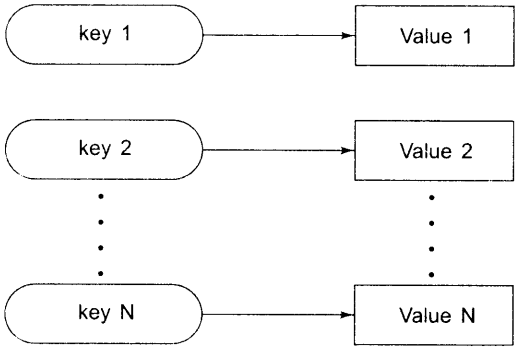

Fig. 14.5 ⇔ *The key-value pairs in a map*

A **map** is commonly called an *associative array*. The key is specified using the subscript operator [ ] as shown below:

```
phone[ "John" ] = 1111;
```

This creates an entry for "John" and associates(i.e. assigns) the value 1111 to it. **phone** is a **map** object. We can change the value, if necessary, as follows:

```
phone[ "John" ] = 9999;
```

This changes the value 1111 to 9999. We can also insert and delete pairs anywhere in the **map** using **insert( )** and **erase( )** functions. Important member functions of the **map** class are listed in Table 14.13.

Table 14.13 *Important member functions of the map class*

Function	*Task*
begin( )	Gives reference to the first element
clear( )	Deletes all elements from the map
empty( )	Decides whether the map is empty or not
end( )	Gives a reference to the end of the map
erase( )	Deletes the specified elements
find( )	Gives the location of the specified element
insert( )	Inserts elements as specified
size( )	Gives the size of the map
swap( )	Exchanges the elements of the given map with those of the invoking map

Program 14.13 shows a simple example of a **map** used as an associative array. Note that **<map>** header must be included.

USING MAPS

```
#include <iostream>
#include <map>
#include <string>

using namespace std;
typedef map<string,int> phoneMap;

int main()
{
        string name;
        int number;
        phoneMap phone;
        cout << "Enter three sets of name and number \n";

        for(int i=0;i<3;i++)
        {
                cin >> name;          // Get key
                cin >> number;        // Get value
                phone[name] = number; // Put them in map
        }

        phone["Jacob"] = 4444;              // Insert Jacob

        phone.insert(pair<string,int> ("Bose", 5555));
        int n = phone.size();
        cout << "\nSize of Map: " << n << "\n\n";
        cout << "List of telephone numbers \n";
        phoneMap::iterator p;
        for(p=phone.begin(); p!=phone.end(); p++)
        {
                cout << (*p).first << "  " << (*p).second << "\n";
        }

        cout << "\n";
        cout << "Enter name: ";             // Get name
        cin >> name;
        number = phone[name];               // Find number
        cout << "Number: " << number << "\n";

        return 0;
}
```

Program 14.3

Output of the Program 14.3 would be:

```
Enter three sets of name and number:
Prasanna 1111
Singh 2222
Raja 3333

Size of Map: 5

List of telephone numbers
Bose 5555

Jacob 4444
Prasanna 1111
Raja 3333
Singh 2222

Enter name: Raja
Number: 3333
```

The program first creates **phone** map interactively with three names and then inserts two more names into the map. Then, it displays all the names and their telephone numbers available in the map. Now the program requests the user to enter the name of a person. The program looks into the map, using the person name as a key, for the associated number and then prints the number.

note

That the names are printed in alphabetical order, although the original data was not. The list is automatically sorted using the key. In our example, the key is the name of person.

We can access the two parts of an entry using the members **first** and **second** with an iterator of the **map** as illustrated in the program. That is,

```
(*p).first
```

gives the key, and

```
(*p).second
```

gives the value.

14.7 Function Objects

A function object is a function that has been wrapped in a class so that it looks like an object. The class has only one member function, the overloaded ( ) operator and no data. The class is templatized so that it can be used with different data types.

Function objects are often used as arguments to certain containers and algorithms. For example, the statement

```
sort(array, array+5, greater<int>());
```

uses the function object **greater<int>( )** to sort the elements contained in **array** in descending order.

Besides comparisons, STL provides many other predefined function objects for performing arithmetical and logical operations as shown in Table 14.14. Note that there are function objects corresponding to all the major C++ operators. For using function objects, we must include **<functional>** header file.

Table 14.14 *STL function objects in <functional>*

Function object	*Type*	*Description*
divides<T>	arithmetic	x/y
equal_to<T>	relational	x == y
greater<T>	relational	x > y
greater_equal<T>	relational	x >= y
less<T>	relational	x < y
less_equal<T>	relational	x <= y
logical_and<T>	logical	x && y
logical_not<T>	logical	!x
logical_or<T>	logical	x || y
minus<T>	arithmetic	x - y
modulus<T>	arithmetic	x % y
negate<T>	arithmetic	- x
not_equal_to<T>	relational	x != y
plus<T>	arithmetic	x + y
multiplies<T>	arithmetic	x * y

Note: The variables x and y represent objects of class T passed to the function object as arguments.

Program 14.4 illustrates the use of the function object **greater<>( )** in **sort( )** algorithm.

USE OF FUNCTION OBJECTS IN ALGORITHMS

```
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;
int main()
{
        int x[] = {10,50,30,40,20};
        int y[] = {70,90,60,80};
```

(Contd)

```
        sort(x,x+5,greater<int>());
        sort(y,y+4);
        for(int i=0; i<5; i++)
                cout << x[i] << " ";
        cout << "\n";
        for(int j=0; j<4; j++)
                cout << y[j] << " ";
        cout << "\n";
        int z[9];
        merge(x,x+5,y,y+4,z);
        for(i=0; i<9; i++)
                cout << z[i] << " ";
        cout << "\n";
        return(0);
}
```

Program 14.4

Output of Program 14.4:

```
50 40 30 20 10
60 70 80 90
50 40 30 20 10 60 70 80 90
```

note

The program creates two arrays **x** and **y** and initializes them with specified values. The program then sorts both of them using the algorithm **sort( )**. Note that **x** is sorted using the function object **greater<int>( )** and **y** is sorted without it and therefore the elements in **x** are in descending order.

The program finally merges both the arrays and displays the content of the merged array. Note the form of **merge()** function and the results it produces.

SUMMARY

- A collection of generic classes and functions is called the Standard Template Library (STL). STL components are part of C++ standard library.
- The STL consists of three main components: containers, algorithms, and iterators.
- Containers are objects that hold data of same type. Containers are divided into three major categories: sequential, associative, and derived.

- Container classes define a large number of functions that can be used to manipulate their contents.
- Algorithms are standalone functions that are used to carry out operations on the contents of containers, such as sorting, searching, copying, and merging.
- Iterators are like pointers. They are used to access the elements of containers thus providing a link between algorithms and containers. Iterators are defined for specific containers and used as arguments to algorithms.
- Certain algorithms use what are known as function objects for some operations. A function object is created by a class that contains only one overloaded operator () function.

Key Terms

- **<algorithm>**
- **<cstdlib>**
- **<deque>**
- **<functional>**
- **<list>**
- **<map>**
- **<numeric>**
- **<queue>**
- **<set>**
- **<string>**
- **<stack>**
- **<vector>**
- algorithms
- associative containers
- bidirectional iterator
- container adaptors
- containers
- **deque**
- derived containers
- forward iterator
- function object
- generic programming
- input iterator
- iterators
- keys
- linear sequence
- list
- **map**
- mapped values
- **multimap**
- multiple keys
- **multiset**
- mutating algorithms
- namespace
- non-mutating algorithms
- numeric algorithms
- output iterator
- priority_queue
- **queue**
- random access iterator
- relational algorithms
- sequence containers
- **set**
- set algorithms

- sorting algorithms
- **stack**
- standard C++ library
- standard template library
- templates
- templatized classes
- tree
- using namespace
- values
- **vector**

Review Questions

14.1 *What is STL? How is it different from the C++ Standard Library? Why is it gaining importance among the programmers?*

14.2 *List the three types of containers.*

14.3 *What is the major difference between a sequence container and an associative container?*

14.4 *What are the best situations for the use of the sequence containers?*

14.5 *What are the best situations for the use of the associative containers?*

14.6 *What is an iterator? What are its characteristics?*

14.7 *What is an algorithm? How STL algorithms are different from the conventional algorithms?*

14.8 *How are the STL algorithms implemented?*

14.9 *Distinguish between the following:*

(a) *lists and vectors*

(b) *sets and maps*

(c) *maps and multimaps*

(d) *queue and deque*

(e) *arrays and vectors*

14.10 *Compare the performance characteristics of the three sequence containers.*

14.11 *Suggest appropriate containers for the following applications:*

(a) *Insertion at the back of a container.*

(b) *Frequent insertions and deletion at both the ends of a container.*

(c) *Frequent insertions and deletions in the middle of a container.*

(d) *Frequent random access of elements.*

14.12 *State whether the following statements are true or false.*

(a) *An iterator is a generalized form of pointer.*

(b) *One purpose of an iterator is to connect algorithms to containers.*

(c) *STL algorithms are member functions of containers.*

(d) *The size of a vector does not change when its elements are removed.*

(e) *STL algorithms can be used with c-like arrays.*

(f) *An iterator can always move forward or backward through a container.*

(g) *The member function* ***end()*** *returns a reference to the last element in the container.*

(h) *The member function* ***back()*** *removes the element at the back of the container.*

(i) *The* ***sort()*** *algorithm requires a random-access iterator.*

(j) *A map can have two or more elements with the same key value.*

Debugging Exercises

14.1 Identify the errror in the following program.

```
#include <iostream.h>
#include <vector>

#define NAMESIZE 40

using namespace std;

class EmployeeMaster
{
private:
    char name[NAMESIZE];
    int id;

public:
    EmployeeMaster()
    {
            strcpy(name, "");
            id = 0;
    }

    EmployeeMaster(char name[NAMESIZE], int id)
            :id(id)
    {
            strcpy(this->name, name);
    }

    EmployeeMaster* getValuesFromUser()
    {
            EmployeeMaster *temp = new EmployeeMaster();
            cout << endl << "Enter user name : ";
            cin >> temp->name;
            cout << endl << "Enter user ID : ";
            cin >> temp->id;
            return temp;
    }
```

```
    void displayRecord()
    {
          cout << endl << "Name : " << name;
          cout << endl << "ID   : " << id << endl;
    }
};

void main()
{
          vector <EmployeeMaster*> emp;
          EmployeeMaster *temp = new EmployeeMaster();
          emp.push_back(getValuesFromUser());
          emp[0]->displayRecord();
          delete temp;

          temp = new EmployeeMaster("AlanKay", 3);
          emp.push_back(temp);
          emp[emp.capacity()]->displayRecord();
          emp[emp.size()]->displayRecord();
}
```

14.2 Identify the error in the following program.

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector <int> v1;
    v1.push_back(10);
    v1.push_back(30);

    vector <int> v2;
    v2.push_back(20);
          v2.push_back(40);

    if(v1==v2)
          cout<<"vectors are equal";
    else
          cout<<"vectors are unequal\t";
          v1.swap(20);
    for(int y=0; y<v1.size(); y++)
```

```
    {
        cout<<"V1="<<v1[y]<<" ";
                cout<<"V2="<<v2[y]<<" ";
    }
    return 0;
}
```

14.3 Identify the error in the following program.

```
#include<iostream>
#include<list>

void main()
{
    list <int> l1;

    l1.push_front(10);
    l1.push_back(20);
    l1.push_front(30);
    l1.push_front(40);
    l1.push_back(10);
    l1.pop_front(40);

    l1.reverse();
    l1.unique();

}
```

Programming Exercises

14.1 *Write a code segment that does the following:*
- (a) *Defines a vector* ***v*** *with a maximum size of 10*
- (b) *Sets the first element of* ***v*** *to 0*
- (c) *Sets the last element of* ***v*** *to 9*
- (d) *Sets the other elements to 1*
- (e) *Displays the contents of* ***v***

14.2 *Write a program using the* ***find()*** *algorithm to locate the position of a specified value in a sequence container.*

14.3 *Write a program using the algorithm* ***count()*** *to count how many elements in a container have a specified value.*

14.4 *Create an array with even numbers and a list with odd numbers. Merge two sequences of numbers into a vector using the algorithm* ***merge()****. Display the vector.*

14.5 *Create a* ***student*** *class that includes a student's first name and his roll_number. Create five objects of this class and store them in a list thus creating a phone_lit. Write a program using this list to display the student name if the roll_number is given and vice-versa.*

14.6 *Redo the Exercise 14.17 using a set.*

14.7 *A table gives a list of car models and the number of units sold in each type in a specified period. Write a program to store this table in a suitable container, and to display interactively the total value of a particular model sold, given the unit-cost of that model.*

14.8 *Write a program that accepts a shopping list of five items from the keyboard and stores them in a vector. Extend the program to accomplish the following:*

(a) *To delete a specified item in the list*

(b) *To add an item at a specified location*

(c) *To add an item at the end*

(d) *To print the contents of the vector*

15

Manipulating Strings

Key Concepts

- C-strings
- The string class
- Creating string objects
- Manipulating strings
- Relational operations on strings
- Comparing strings
- String characteristics
- Swapping strings

15.1 Introduction

A string is a sequence of characters. We know that C++ does not support a built-in string type. We have used earlier null-terminated character arrays to store and manipulate strings. These strings are called *C-strings* or *C-style strings*. Operations on C-strings often become complex and inefficient. We can also define our own string classes with appropriate member functions to manipulate strings. This was illustrated in Program 7.4 (Mathematical Operation of Strings).

ANSI standard C++ now provides a new class called **string**. This class improves on the conventional C-strings in several ways. In many situations, the string objects may be used like any other built-in type data. Further, although it is not considered as a part of the STL, **string** is treated as another container class by C++ and therefore all the algorithms that are applicable for containers can be used with the **string** objects. For using the **string** class, we must include <**string**> in our program.

The **string** class is very large and includes many constructors, member functions and operators. We may use the constructors, member functions and operators to achieve the following:

- Creating string objects
- Reading string objects from keyboard
- Displaying string objects to the screen
- Finding a substring from a string
- Modifying string objects
- Comparing string objects
- Adding string objects
- Accessing characters in a string
- Obtaining the size of strings
- And many other operations

Table 15.1 gives prototypes of three most commonly used constructors and Table 15.2 gives a list of important member functions. Table 15.3 lists a number of operators that can be used on **string** objects.

Table 15.1 *Commonly used string constructors*

Constructor	*Usage*
String();	For creating an empty string
String(const chat *str);	For creating a string object from a null-terminated string
String(const string & str);	For creating a string object from other string object

Table 15.2 *Important functions supported by the string class*

Function	*Task*
append()	Appends a part of string to another string
Assign()	Assigns a partial string
at()	Obtains the character stored at a specified location
Begin()	Returns a reference to the start of a string
capacity()	Gives the total elements that can be stored.
compare()	Compares string against the invoking string
empty()	Returns true if the string is empty; Otherwise returns false
end()	Returns a reference to the end of a string
erase()	Removes characters as specified
find()	Searches for the occurrence of a specified substring
insert()	Inserts characters at a specified location
length()	Gives the number of elements in a string
max size()	Gives the maximum possible size of a string object in a give system
replace()	Replace specified characters with a given string
resize()	Changes the size of the string as specified
size()	Gives the number of characters in the string
swap()	Swaps the given string with the invoking string

Table 15.3 *Operators for string objects*

Operator	*Meaning*
=	Assignment
+	Concatenation
+=	Concatenation assignment
==	Equality
!=	Inequality
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
[]	Subscription
<<	Output
>>	Input

15.2 Creating (string) Objects

We can create **string** objects in a number of ways as illustrated below:

```
string s1;              // Using constructor with no argument
string s2("xyz");       // Using one-argument constructor
s1 = s2;                // Assigning string objects
s3 = "abc" + s2         // Concatenating strings
cin >> s1;              // Reading through keyboard (one word)
getline(cin, s1);       // Reading through keyboard a line of text
```

The overloaded + operator concatenates two string objects. We can also use the operator += to append a string to the end of a string. Examples:

```
s3 += s1;         // s3 = s3 + s1
s3 += "abc";      // s3 = s3 + "abc"
```

The operators << and >> are overloaded to handle input and output of string objects. Examples:

```
cin >> s2;              // Input to string object (one word)
cout << s2;             // Displays the contents of s2
getline(cin, s2);       // Reads embedded blanks
```

note

Using **cin** and >> operator we can read only one word of a string while the **getline()** function permits us to read a line of text containing embedded blanks.

Program 15.1 demonstrates the several ways of creating string objects in a program.

CREATING STRING OBJECTS

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
     // Creating string objects
     string s1;                    // Empty string object
     string s2(" New");            // Using string constant
     string s3(" Delhi");

     // Assigning value to string objects
     s1 = s2;                      // Using string object
     cout << "S1 = " << s1 << "\n";

     // Using a string constant
     s1 = "Standard C++";
     cout << "Now S1 = " << s1 << "\n";

     // Using another object
     string s4(s1);
     cout << "S4 = " << s4 << "\n\n";

     // Reading through keyboard
     cout << "ENTER A STRING \n";
     cin >> s4;                          // Delimited by blank space
     cout << "Now S4 = " << s4 << "\n\n";

     // Concatenating strings
     s1 = s2 + s3;
     cout << "S1 finally contains: " << s1 << "\n";

     return 0;
}
```

PROGRAM 15.1

The output of Program 15.1 would be:

```
S1 =  New
Now S1 = Standard C++
S4 = Standard C++
```

```
ENTER A STRING
COMPUTER CENTRE
Now S4 = COMPUTER

S1 finally contains:  New Delhi
```

15.3 Manipulating String Objects

We can modify contents of **string** objects in several ways, using the member functions such as **insert(), replace(), erase(),** and **append()**. Program 15.2 demonstrates the use of some of these functions.

MODIFYING STRING OBJECTS

```
#include <iostream
#include <string>

using namespace std;

int main()
{
  string s1("12345");
  string s2("abcde");

  cout << "Original Strings are: \n";
  cout << "S1: " << s1 << "\n";
  cout << "S2: " << s2 << "\n\n";

  // Inserting a string into another
  cout << "Place S2 inside S1 \n";
  s1.insert(4,s2);
  cout << "Modified S1: " << s1 << "\n\n";

  // Removing characters in a string
  cout << "Remove 5 Characters from S1 \n";
  s1.erase(4,5);
  cout << "Now S1: " << s1 << "\n\n";

  // Replacing characters in a string
  cout << "Replace Middle 3 Characters in S2 with S1 \n";
```

(Contd)

```
    s2.replace(1,3,s1);
    cout << "Now S2: " << s2 << "\n";

    return 0;
}
```

PROGRAM 15.2

The output of Program 15.2 given below illustrates how strings are manipulated using string functions.

```
Original Strings are:
S1: 12345
S2: abcde

Place S2 inside S1
Modified S1: 1234abcde5

Remove 5 Characters from S1
Now S1: 12345

Replace Middle 3 Characters in S2 with S1
Now S2: a12345e
```

note

Analyse how arguments of each function used in this program are implemented.

15.4 Relational Operations

A number of operators that can be used on strings are defined for **string** objects (Table 15.3). We have used in the earlier examples the operators = and + for creating objects. We can also apply the relational operators listed in Table 15.3. These operators are overloaded and can be used to compare **string** objects. The **compare()** function can also be used for this purpose.

RELATIONAL OPERATIONS ON STRING OBJECTS

```
#include <iostream>
#include <string>

using namespace std;
```

(Contd)

```
int main()
{
  string s1("ABC");
  string s2("XYZ");
  string s3 = s1 + s2;

  if(s1 != s2)
        cout << "s1 is not equal to s2 \n";
  if(s1 > s2)
        cout << "s1 greater than s2 \n";
  else
        cout << "s2 greater than s1 \n";
  if(s3 == s1 + s2)
        cout << "s3 is equal to s1+s2 \n\n";
  int x = s1.compare(s2);
  if(x == 0)
        cout << "s1 == s2 \n";
  else if(x > 0)
        cout << "s1 > s2 \n";
  else                          // x < 0
        cout << "s1 < s2 \n";

  return 0;
}
```

PROGRAM 15.3

Program 15.3 shows how these operators are used.

This program produces the following output:

```
s1 is not equal to s2
s2 greater than s1
s3 is equal to s1+s2

s1 < s2
```

15.5 String Characteristics

Class **string** supports many functions that could be used to obtain the characteristics of strings such as size, length, capacity, etc. The size or length denotes the number of elements

currently stored in a given string. The capacity indicates the total elements that can be stored in the given string. Another characteristic is the *maximum size* which is the largest possible size of a string object that the given system can support. Program 15.4 illustrates how these characteristics are obtained and used in an application.

OBTAINING STRING CHARACTERISTICS

```
#include <iostream>
#include <string>

using namespace std;

void display(string &str)
{
   cout << "Size = " << str.size() << "\n";
   cout << "Length = " << str.length() << "\n";
   cout << "Capacity = " << str.capacity() << "\n";
   cout << "Maximum Size = " << str.max_size() << "\n";
   cout << "Empty: " << (str.empty() ? "yes" : "no");
   cout << "\n\n";
}

int main()
{
   string str1;

   cout << "Initial status: \n";
   display(str1);

   cout << "Enter a string (one word) \n";
   cin >> str1;
   cout  << "Status now: \n";
   display(str1);

   str1.resize(15);
   cout << "Status after resizing: \n";
   display(str1);
   cout << "\n";

   return 0;
}
```

PROGRAM 15.4

Shown below is the output of Program 15.4:

```
Initial status:
Size = 0
```

```
Length = 0
Capacity = 0
Maximum Size = 4294967293
Empty: yes

Enter a string (one word)
INDIA
Status now:
Size = 5
Length = 5
Capacity = 31
Maximum Size = 4294967293
Empty: no

Status after resizing:
Size = 15
Length = 15
Capacity = 31
Maximum Size = 4294967293
Empty: no
```

The size and length of 0 indicate that the string **str1** contains no characters. The size and length are always the same. The **str1** has a capacity of zero initially but its capacity has increased to 31 when a string is assigned to it. The maximum size of a string in this system is 4294967293. The function **empty()** returns **true** if **str1** is empty; otherwise **false**.

15.6 Accessing Characters in Strings

We can access substrings and individual characters of a string in several ways. The **string** class supports the following functions for this purpose:

`at()`	for accessing individual characters
`substr()`	for retrieving a substring
`find()`	for finding a specified substring
`find_first_of()`	for finding the location of first occurrence of the specified character(s)
`find_last_of()`	for finding the location of last occurrence of the specified character(s)

We can also use the overloaded [ ] operator (which makes a **string** object look like an array) to access individual elements in a string. Program 15.5 demonstrates the use of some of these functions.

ACCESSING AND MANIPULATING CHARACTERS

```
#include <iostream>
#include <string>
```

(Contd)

```
using namespace std;

int main()
{
  string s("ONE TWO THREE FOUR");

  cout << "The string contains: \n";
  for(int i=0;i<s.length();i++)
        cout << s.at(i);            // Display one character
  cout << "\nString is shown again: \n";
  for(int j=0;j<s.length();j++)
        cout << s[j];

  int x1 = s.find("TWO");
  cout << "\n\nTWO is found at: " << x1 << "\n";

  int x2 = s.find_first_of('T');
  cout << "\nT is found first at: " << x2 << "\n";
  int x3 = s.find_last_of('R');
  cout << "\nR is last found at: " << x3 << "\n";

  cout << "\nRetrieve and print substring TWO \n";

  cout << s.substr(x1,3);
  cout << "\n";

  return 0;
}
```

PROGRAM 15.5

Shown below is the output of Program 15.5:

```
The string contains:
ONE TWO THREE FOUR
String is shown again:
ONE TWO THREE FOUR

TWO is found at: 4

T is found first at: 4

R is last fount at: 17

Retrieve and print substring TWO
TWO
```

The statement

```
int x = s1.compare(s2);
```

compares the string **s1** against **s2** and **x** is assigned 0 if the strings are equal, a positive number if **s1** is *lexicographically* greater than **s2** or a negative number otherwise.

The statement

```
int a = s1.compare(0,2,s2,0,2);
```

compares portions of **s1** and **s2**. The first two arguments give the starting subscript and length of the portion of **s1** to compare to **s2**, that is supplied as the third argument. The fourth and fifth arguments specify the starting subscript and length of the portion of **s2** to be compared. The value assigned to **a** is 0, if they are equal, 1 if substring of **s1** is greater than the substring of **s2**, –1 otherwise.

The statement

```
s2.swap(s2);
```

exchanges the contents of the strings **s1** and **s2**.

SUMMARY

- Manipulation and use of C-style strings become complex and inefficient. ANSI C++ provides a new class called **string** to overcome the deficiencies of C-strings.
- The string class supports many constructors, member functions and operators for creating and manipulating string objects. We can perform the following operations on the strings:
  - Reading strings from keyboard
  - Assigning strings to one another
  - Finding substrings
  - Modifying strings
  - Comparing strings and substrings
  - Accessing characters in strings]
  - Obtaining size and capacity of strings
  - Swapping strings
  - Sorting strings

Key Terms

- **<string>**
- **append()**
- **assign()**
- **at()**
- **begin()**
- capacity
- **capacity()**
- **compare()**
- comparing strings
- C-strings
- C-style strings
- **empty()**
- **end()**
- **erase()**
- **find()**
- **find_first_of()**
- **find_last_of()**
- **getline()**
- **insert()**
- length
- **length()**
- lexicographical
- **max_size()**
- maximum size
- relational operators
- **replace()**
- size
- **size()**
- string
- string class
- string constructors
- string objects
- **substr()**
- substring
- **swap()**
- swapping strings

Review Questions

15.1 *State whether the following statements are TRUE or FALSE:*

(a) *For using **string** class, we must include the header <string>.*

(b) ***string** objects are null terminated.*

(c) *The elements of a **string** object are numbered from 0.*

(d) *Objects of **string** class can be copied using the assignment operator.*

(e) *Function **end()** returns an iterator to the invoking **string** object.*

15.2 *How does a **string** type string differ from a C-type string?*

15.3 *The following statements are available to read strings from the keyboard.*

```
(a)  cin >> s1;
(b)  getline(cin, s1);
```

*where **s1** is a **string** object. Distinguish their behaviour.*

15.4 *Consider the following segment of a program:*

```
string s1("man"), s2, s3;
s2.assign(s1);
s3 = s1;
string s4("wo" + s1);
s2 += "age";
s3.append("ager");
s1[0] = 'v';
```

State the contents of the objects ***s1***, ***s2***, ***s3*** *and* ***s4*** *when executed.*

15.5 *We can access string elements using*

(a) ***at()*** *function*

(b) *subscript operator [ ]*

Compare their behaviour.

15.6 *What does each of the following statements do?*

(a) `s.replace(n,1,"/");`

(b) `s.erase(10);`

(c) `s1.insert(10,s2);`

(d) `int x = s1.compare(0, s2.size(), s2);`

(e) `s2 = s1.substr(10, 5);`

15.7 *Distinguish between the following pair of functions.*

(a) *max_size() and capacity()*

(b) *find() and rfind()*

(c) *begin() and rbegin()*

Debugging Exercises

15.1 Identify the error in the following program.

```
#include <iostream.h>
#include <string>

using namespace std;

void main()
{
        string str1("ghi");
        string str2("abc" + "def");
        str2+=str1;
        cout << str2.c_str();
}
```

15.2 Identify the error in the following program.

```
#include <iostream.h>
```

```
#include <string>

using namespace std;
void main()
{
        string str1("ABCDEF");
        string str2("123");
        string str3;

        str1.insert(2, str2);
        str1.erase(2,2);
        str1.replace(2,str2);

        cout << str1.c_str();
        cout << endl;
}
```

15.3 Identify the error in the following program.

```
#include <iostream>
#include <string>

using namespace std;

class Product
{
        int iProductNumber;
        string strProductName;
public:
      Product()
      {
      }

      Product(const int &number, const string &name)
      {
            setProductNumber(number);
            setProductName(name);
      }

      void setProductNumber(int n)
      {
             iProductNumber = n;
```

```
        }

        void setProductName(const string str)
        {
                strProductName = str;
        }

        int getProductNumber()
        {
                return iProductNumber;
        }

        const string getProductName()
        {
                return strProductName ;
        }

        Product& operator = (Product &source)
        {
                setProductNumber(source.iProductNumber);
                string strTemp = source.strProductName;
                setProductName(strTemp);
                return *this;
        }

        void display()
        {
                cout << "ProductName : " << getProductName();
                cout << " " ;
                cout << "ProductNumber : " << getProductNumber();
                cout << endl;
        }
};

void main()
{
        Product p1(1, 5);
        Product p2(3, "Dates");
        Product p3;
        p3 = p2 = p1;
```

```
        p3.display();
        p2.display();
    }
```

15.4 Find errors, if any, in the following segment of code.

```
int len = s1. length();
for (int i=0; 1<len;++i)
        cout << s1.at[];
```

Programming Exercises

15.1 *Write a program that reads the name*

```
Martin Luther King
```

from the keyboard into three separate ***string*** *objects and then concatenates them into a new* ***string*** *object using*

(a) + *operator and*

(b) ***append()*** *function.*

15.2 *Write a program using an iterator and* ***while()*** *construct to display the contents of a* **string** object.

15.3 *Write a program that reads several city names from the keyboard and displays only those names beginning with characters "B" or "C".*

15.4 *Write a program that will read a line of text containing more than three words and then replace all the blank spaces with an underscore(_).*

15.5 *Write a program that counts the number of occurrences of a particular character, say 'e', in a line of text.*

15.6 *Write a program that reads the following text and counts the number of times the word "It" appears in it.*

```
It is new. It is singular.
It is simple. It must succeed!
```

15.7 *Modify the program in Exercise 15.14 to count the number of words which start with the character 's'.*

15.8 *Write a program that reads a list of countries in random order and displays them in alphabetical order. Use comparison operators and functions.*

15.9 *Given a string*

```
string s("123456789");
```

Write a program that displays the following:

1
2 3 2
3 4 5 4 3
4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5

16

New Features of ANSI C++ Standard

Key Concepts

- Boolean type data
- Wide-character literals
- Constant casting
- Static casting
- Dynamic casting
- Reinterpret casting
- Runtime type information
- Explicit constructors
- Mutable member data
- Namespaces
- Nesting of namespaces
- Operator keywords
- Using new keywords
- New style for headers

16.1 Introduction

The ISO / ANSI C++ Standard adds several new features to the original C++ specifications. Some are added to provide better control in certain situations and others are added for providing conveniences to C++ programmers. It is therefore important to note that it is technically possible to write full-fledged programs without using any of the new features. Important features added are:

1. New data types
   - bool
   - wchar_t
2. New operators
   - const_cast
   - static_cast
   - dynamic_cast
   - reinterpret_cast
   - typeid
3. Class implementation
   - Explicit constructors
   - Mutable members
4. Namespace scope

5. Operator keywords
6. New keywords
7. New headers

We present here a brief overview of these features.

16.2 New Data Types

The ANSI C++ has added two new data types to enhance the range of data types available in C++. They are **bool** and **wchar_t**.

The bool Data Type

The data type **bool** has been added to hold a Boolean value, **true** or **false**. The values **true** and **false** have been added as keywords to the C++ language. The **bool** type variables can be declared as follows.

```
bool b1;                    // declare b1 as bool type
b1 = true;                  // assign true value to it
bool b2 = false;            // declare and initialize
```

The default numeric value of **true** is 1 and **false** is 0. Therefore, the statements

```
cout << "b1 = " << b1;      // b1 is true
cout << "b2 = " << b2;      // b2 is false
```

will display the following output:

```
b1 = 1
b2 = 0
```

We can use the **bool** type variables or the values **true** and **false** in mathematical expressions. For instance,

```
int x = false + 5*m - b1;
```

is valid and the expression on the right evaluates to 9 assuming **b1** is true and **m** is 2. Values of type **bool** are automatically elevated to integers when used in non-Boolean expressions.

It is possible to convert implicitly the data types pointers, integers or floating point values to **bool** type. For example, the statements

```
bool x  = 0;
bool y = 100;
bool z = 15.75;
```

assign **false** to **x** and **true** to **y** and **z**.

Program 16.1 demonstrates the features of **bool** type data.

USE OF bool TYPE DATA

```
#include <iostream>

using namespace std;

int main()
{
        int x1 = 10,x2 = 20,m = 2;
        bool b1, b2;

        b1 = x1 == x2;      // False
        b2 = x1 < x2;               // True

        cout << "b1 is " << b1 << "\n";
        cout << "b2 is " << b2 << "\n";

        bool b3  =  true;
        cout << "b3 is " << b3 << "\n";

        if(b3)
              cout << "Very Good" << "\n";
        else
              cout << "Very Bad" << "\n";

        int x3 = false + 5*m-b3;
        b1 = x3;
        b2 = 0;
        cout << "x3 = " << x3 << "\n";
        cout << "Now b1 = " << b1 << " and b2 = " << b2 << "\n";

        return 0;
}
```

PROGRAM 16.1

The output of Program 16.1 would be:

```
b1 is 0
b2 is 1
```

```
b3 is 1
Very Good
x3 = 9
Now b1 = 1 and b2 = 0
```

The wchar_t Data Type

The character type **wchar_t** has been defined in ANSI C++ to hold 16-bit wide characters. The 16-bit characters are used to represent the character sets of languages that have more than 255 characters, such as Japanese. This is important if we are writing programs for international distribution.

ANSI C++ also introduces a new character literal known as *wide_character* literal which uses two bytes of memory. Wide_character literals begin with the letter L, as follows:

```
L'xy'     // wide_character literal
```

16.3 New Operators

We have used cast operators (also known as *casts* or *type casts*) earlier in a number of programs. As we know, casts are used to convert a value from one type to another. This is necessary in situations where automatic conversions are not possible. We have used the following forms of casting:

```
double x = double(m);          // C++ type casting
double y = (double)n;          // C-type casting
```

Although these casts still work, ANSI C++ has added several new cast operators known as *static casts*, *dynamic casts*, *reinterpret casts* and *constant casts*. It also adds another operator known as **typeid** to verify the types of unknown objects.

The static_cast Operator

Like the conventional cast operators, the **static_cast** operator is used for any standard conversion of data types. It can also be used to cast a base class pointer into a derived class pointer. Its general form is:

```
static_cast<type>(object)
```

Here, *type* specifies the target type of the cast, and *object* is the object being cast into the new type. Examples:

```
int m = 10;
double x = static_cast<double> (m);
char ch = static_cast<char> (m);
```

The first statement casts the variable **m** to type **double** and the second casts it to type **char**.

Why use this new type when the old style still works? The syntax of the old one blends into the rest of the lines and therefore it is difficult to locate them. The new format is easy to spot and to search for using automated tools.

The const_cast Operator

The **const_cast** operator is used to explicitly override **const** or **volatile** in a cast. It takes the form

```
const_cast<type>(object)
```

Since the purpose of this operator is to change its **const** or **volatile** attributes, the *target type* must be the same as the *source type*. It is mostly used for removing the const_ness of an object.

The reinterpret_cast Operator

The **reinterpret_cast** is used to change one type into a fundamentally different type. For example, it can be used to change a pointer type object to integer type object or vice versa. It takes the following form:

```
reinterpret_cast<type>(object)
```

This operator should be used for casting inherently incompatible types. Examples:

```
int m;
float x;
int* intptr;
float* floatptr;
intptr = reinterpret_cast<int*> (m);
floatptr = reinterpret_cast<float*> (x);
```

The dynamic_cast Operator

The dynamic cast is used to cast the type of an object at runtime. Its main application is to perform casts on polymorphic objects. Recall that polymorphic objects are created by base classes that contain virtual functions. It takes the form:

```
dynamic_cast<type>(object)
```

The *object* is a base class object whose type is to be checked and casted. It casts the type of object to *type*. It never performs an invalid conversion. It checks that the conversion is legal at runtime. If the conversion is not valid, it returns NULL.