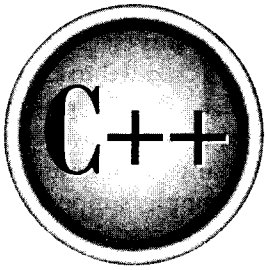
The Complete Reference



Chapter 24

Introducing the Standard Template Library

This chapter explores what is considered by many to be the most important feature added to C++ in recent years: the *standard template library* (*STL*). The inclusion of the STL was one of the major efforts that took place during the standardization of C++. It provides general-purpose, templatized classes and functions that implement many popular and commonly used algorithms and data structures, including, for example, support for vectors, lists, queues, and stacks. It also defines various routines that access them. Because the STL is constructed from template classes, the algorithms and data structures can be applied to nearly any type of data.

The STL is a complex piece of software engineering that uses some of C++'s most sophisticated features. To understand and use the STL, you must have a complete understanding of the C++ language, including pointers, references, and templates. Frankly, the template syntax that describes the STL can seem quite intimidating—although it looks more complicated than it actually is. While there is nothing in this chapter that is any more difficult than the material in the rest of this book, don't be surprised or dismayed if you find the STL confusing at first. Just be patient, study the examples, and don't let the unfamiliar syntax override the STL's basic simplicity.

The purpose of this chapter is to present an overview of the STL, including its design philosophy, organization, constituents, and the programming techniques needed to use it. Because the STL is a large library, it is not possible to discuss all of its features here. However, a complete reference to the STL is provided in Part Four.

This chapter also describes one of C++'s most important classes: **string**. The **string** class defines a string data type that allows you to work with character strings much as you do other data types: using operators. The **string** class is closely related to the STL.

An Overview of the STL

Although the standard template library is large and its syntax can be intimidating, it is actually quite easy to use once you understand how it is constructed and what elements it employs. Therefore, before looking at any code examples, an overview of the STL is warranted.

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 act, more or less, like 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.

When referring to the various iterator types in template descriptions, this book will use the following terms:

Term	Represents
BiIter	Bidirectional iterator
ForIter	Forward iterator
InIter	Input iterator
OutIter	Output iterator
RandIter	Random access iterator

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. These functions return true/false results. But the precise conditions that make them return true or false are defined by you. For the rest of this chapter, when a unary predicate function is required, it will be notated using the type **UnPred**. When a binary predicate is required, the type **BinPred** will be used. In a binary predicate, the arguments are always in the order of *first,second*. For both unary and binary predicates, the arguments will contain values of the type of objects being stored by the container.

Some algorithms and classes use a special type of binary predicate that compares two elements. Comparison functions return true if their first argument is less than their second. Comparison functions will be notated using the type **Comp**.

In addition to the headers required by the various STL classes, the C++ standard library includes the **<utility>** and **<functional>** headers, which provide support for the STL. For example, the template class **pair**, which can hold a pair of values, is defined in **<utility>**. We will make use of **pair** later in this chapter.

The templates in **<functional>** help you construct objects that define **operator( )**. These are called *function objects* and they may be used in place of function pointers in many places. There are several predefined function objects declared within **<functional>**. They are shown here:

plus	minus	multiplies	divides	modulus
negate	equal_to	not_equal_to	greater	greater_equal
less	less_equal	logical_and	logical_or	logical_not

Perhaps the most widely used function object is **less**, which determines when one object is less than another. Function objects can be used in place of actual function pointers in the STL algorithms described later. Using function objects rather than function pointers allows the STL to generate more efficient code.

Two other entities that populate the STL are *binders* and *negators*. A binder binds an argument to a function object. A negator returns the complement of a predicate.

One final term to know is *adaptor*. In STL terms, an adaptor transforms one thing into another. For example, the container **queue** (which creates a standard queue) is an adaptor for the **deque** container.

The Container Classes

As explained, containers are the STL objects that actually store data. The containers defined by the STL are shown in Table 24-1. Also shown are the headers necessary to use each container. The **string** class, which manages character strings, is also a container, but it is discussed later in this chapter.

Container	Description	Required Header
bitset	A set of bits.	<bitset>
deque	A double-ended queue.	<deque>
list	A linear list.	<list>
map	Stores key/value pairs in which each key is associated with only one value.	<map>
multimap	Stores key/value pairs in which one key may be associated with two or more values.	<map>
multiset	A set in which each element is not necessarily unique.	<set>
priority_queue	A priority queue.	<queue>
queue	A queue.	<queue>
set	A set in which each element is unique.	<set>
stack	A stack.	<stack>
vector	A dynamic array.	<vector>

Table 24-1. *The Containers Defined by the STL*

Since the names of the generic placeholder types in a template class declaration are arbitrary, the container classes declare **typedef**ed versions of these types. This makes the type names concrete. Some of the most common **typedef** names are shown here:

size_type	Some type of integer
reference	A reference to an element
const_reference	A **const** reference to an element
iterator	An iterator
const_iterator	A **const** iterator
reverse_iterator	A reverse iterator
const_reverse_iterator	A **const** reverse iterator
value_type	The type of a value stored in a container
allocator_type	The type of the allocator
key_type	The type of a key
key_compare	The type of a function that compares two keys
value_compare	The type of a function that compares two values

General Theory of Operation

Although the internal operation of the STL is highly sophisticated, to use the STL is actually quite easy. First, you must decide on the type of container that you wish to use. Each offers certain benefits and trade-offs. For example, a **vector** is very good when a random-access, array-like object is required and not too many insertions or deletions are needed. A **list** offers low-cost insertion and deletion but trades away speed. A **map** provides an associative container, but of course incurs additional overhead.

Once you have chosen a container, you will use its member functions to add elements to the container, access or modify those elements, and delete elements. Except for **bitset**, a container will automatically grow as needed when elements are added to it and shrink when elements are removed.

Elements can be added to and removed from a container a number of different ways. For example, both the sequence containers (**vector**, **list**, and **deque**) and the associative containers (**map**, **multimap**, **set**, and **multiset**) provide a member function called **insert( )**, which inserts elements into a container, and **erase( )**, which removes elements from a container. The sequence containers also provide **push_back( )** and **pop_back( )**, which add an element to or remove an element from the end, respectively. These functions are probably the most common way that individual elements are added to or removed from a sequence container. The **list** and **deque**

containers also include **push_front( )** and **pop_front( )**, which add and remove elements from the start of the container.

One of the most common ways to access the elements within a container is through an iterator. The sequence and the associative containers provide the member functions **begin( )** and **end( )**, which return iterators to the start and end of the container, respectively. These iterators are very useful when accessing the contents of a container. For example, to cycle through a container, you can obtain an iterator to its beginning using **begin( )** and then increment that iterator until its value is equal to **end( )**.

The associative containers provide the function **find( )**, which is used to locate an element in an associative container given its key. Since associative containers link a key with its value, **find( )** is how most elements in such a container are located.

Since a **vector** is a dynamic array, it also supports the standard array-indexing syntax for accessing its elements.

Once you have a container that holds information, it can be manipulated using one or more algorithms. The algorithms not only allow you to alter the contents of a container in some prescribed fashion, but they also let you transform one type of sequence into another.

In the following sections, you will learn to apply these general techniques to three representative containers: **vector**, **list**, and **map**. Once you understand how these containers work, you will have no trouble using the others.

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*.

For maximum flexibility and portability, any object that will be stored in a **vector** should define a default constructor. It should 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.

Several of the member functions defined by **vector** are shown in Table 24-2. (Remember, Part Four contains a complete reference to the STL classes.) Some of the most commonly used member functions are **size( )**, **begin( )**, **end( )**, **push_back( )**, **insert( )**, and **erase( )**. The **size( )** function returns the current size of the vector. This function is quite useful because it allows you to determine the size of a vector at run time. Remember, vectors will increase in size as needed, so the size of a vector must be determined during execution, not during compilation.

The **begin( )** function returns an iterator to the start of the vector. The **end( )** function returns an iterator to the end of the vector. As explained, iterators are similar to pointers, and it is through the use of the **begin( )** and **end( )** functions that you obtain an iterator to the beginning and end of a vector.

The **push_back( )** function puts a value onto the end of the vector. If necessary, the vector is increased in length to accommodate the new element. You can also add elements to the middle using **insert( )**. A vector can also be initialized. In any event, once a vector contains elements, you can use array subscripting to access or modify those elements. You can remove elements from a vector using **erase( )**.

Member	Description
reference back( ); const_reference back( ) const;	Returns a reference to the last element in the vector.
iterator begin( ); const_iterator begin( ) const;	Returns an iterator to the first element in the vector.
void clear( );	Removes all elements from the vector.
bool empty( ) const;	Returns true if the invoking vector is empty and false otherwise.
iterator end( ); const_iterator end( ) const;	Returns an iterator to the end of the vector.
iterator erase(iterator *i*);	Removes the element pointed to by *i*. Returns an iterator to the element after the one removed.
iterator erase(iterator *start*, iterator *end*);	Removes the elements in the range *start* to *end*. Returns an iterator to the element after the last element removed.
reference front( ); const_reference front( ) const;	Returns a reference to the first element in the vector.
iterator insert(iterator *i*, const T &*val*);	Inserts *val* immediately before the element specified by *i*. An iterator to the element is returned.
void insert(iterator *i*, size_type *num*, const T & *val*)	Inserts *num* copies of *val* immediately before the element specified by *i*.
template <class InIter> void insert(iterator *i*, InIter *start*, InIter *end*);	Inserts the sequence defined by *start* and *end* immediately before the element specified by *i*.
reference operator[ ](size_type *i*) const; const_reference operator[ ](size_type *i*) const;	Returns a reference to the element specified by *i*.
void pop_back( );	Removes the last element in the vector.
void push_back(const T &*val*);	Adds an element with the value specified by *val* to the end of the vector.
size_type size( ) const;	Returns the number of elements currently in the vector.

Table 24-2. *Some Commonly Used Member Functions Defined by* ***vector***

Here is a short example that illustrates the basic operation of a vector.

```
// Demonstrate a vector.
#include <iostream>
#include <vector>
#include <cctype>
using namespace std;

int main()
{
  vector<char> v(10); // create a vector of length 10
  unsigned 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;
```

```
  return 0;
}
```

The output of this program is shown here:

```
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
```

Let's look at this program carefully. In **main( )**, a character vector called **v** is created with an initial capacity of 10. That is, **v** initially contains 10 elements. This is confirmed by calling the **size( )** member function. Next, these 10 elements are initialized to the characters a through j and the contents of **v** are displayed. Notice that the standard array subscripting notation is employed. Next, 10 more elements are added to the end of **v** using the **push_back( )** function. This causes **v** to grow in order to accommodate the new elements. As the output shows, its size after these additions is 20. Finally, the values of **v**'s elements are altered using standard subscripting notation.

There is one other point of interest in this program. Notice that the loops that display the contents of **v** use as their target value **v.size( )**. One of the advantages that vectors have over arrays is that it is possible to find the current size of a vector. As you can imagine, this can be quite useful in a variety of situations.

Accessing a Vector Through an Iterator

As you know, arrays and pointers are tightly linked in C++. An array can be accessed either through subscripting or through a pointer. The parallel to this in the STL is the link between vectors and iterators. You can access the members of a vector using subscripting or through the use of an iterator. The following example shows how.

```
// Access the elements of a vector through an iterator.
#include <iostream>
#include <vector>
#include <cctype>
```

```
using namespace std;

int main()
{
  vector<char> v(10); // create a vector of length 10
  vector<char>::iterator p; // create an iterator
  int i;

  // assign elements in vector a value
  p = v.begin();
  i = 0;
  while(p != v.end()) {
    *p = i + 'a';
    p++;
    i++;
  }

  // display contents of vector
  cout << "Original contents:\n";
  p = v.begin();
  while(p != v.end()) {
    cout << *p << " ";
    p++;
  }
  cout << "\n\n";

  // change contents of vector
  p = v.begin();
  while(p != v.end()) {
    *p = toupper(*p);
    p++;
  }

  // display contents of vector
  cout << "Modified Contents:\n";
  p = v.begin();
  while(p != v.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl;
```

```
  return 0;
}
```

The output from this program is

```
Original contents:
a b c d e f g h i j

Modified Contents:
A B C D E F G H I J
```

In the program, notice how the iterator **p** is declared. The type **iterator** is defined by the container classes. Thus, to obtain an iterator for a particular container, you will use a declaration similar to that shown in the example: simply qualify **iterator** with the name of the container. In the program, **p** is initialized to point to the start of the vector by using the **begin( )** member function. This function returns an iterator to the start of the vector. This iterator can then be used to access the vector an element at a time by incrementing it as needed. This process is directly parallel to the way a pointer can be used to access the elements of an array. To determine when the end of the vector has been reached, the **end( )** member function is employed. This function returns an iterator to the location that is one past the last element in the vector. Thus, when **p** equals **v.end( )**, the end of the vector has been reached.

Inserting and Deleting Elements in a Vector

In addition to putting new values on the end of a vector, you can insert elements into the middle using the **insert( )** function. You can also remove elements using **erase( )**. The following program demonstrates **insert( )** and **erase( )**.

```
// Demonstrate insert and erase.
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<char> v(10);
  vector<char> v2;
  char str[] = "<Vector>";
  unsigned int i;
```

```
// initialize v
for(i=0; i<10; i++) v[i] = i + 'a';

// copy characters in str into v2
for(i=0; str[i]; i++) v2.push_back(str[i]);

// display original contents of vector
cout << "Original contents of v:\n";
for(i=0; i<v.size(); i++) cout << v[i] << " ";
cout << "\n\n";

vector<char>::iterator p = v.begin();
p += 2; // point to 3rd element

// insert 10 X's into v
v.insert(p, 10, 'X');

// display contents after insertion
cout << "Size after inserting X's = " << v.size() << endl;
cout << "Contents after insert:\n";
for(i=0; i<v.size(); i++) cout << v[i] << " ";
cout << "\n\n";

// remove those elements
p = v.begin();
p += 2; // point to 3rd element
v.erase(p, p+10); // remove next 10 elements

// display contents after deletion
cout << "Size after erase = " << v.size() << endl;
cout << "Contents after erase:\n";
for(i=0; i<v.size(); i++) cout << v[i] << " ";
cout << "\n\n";

// Insert v2 into v
v.insert(p, v2.begin(), v2.end());
cout << "Size after v2's insertion = ";
cout << v.size() << endl;
cout << "Contents after insert:\n";
for(i=0; i<v.size(); i++) cout << v[i] << " ";
cout << endl;
```

```
  return 0;
}
```

This program produces the following output:

```
Original contents of v:
a b c d e f g h i j

Size after inserting X's = 20
Contents after insert:
a b X X X X X X X X X c d e f g h i j

Size after erase = 10
Contents after erase:
a b c d e f g h i j

Size after v2's insertion = 18
Contents after insert:
a b < V e c t o r > c d e f g h i j
```

This program demonstrates two forms of **insert( )**. The first time it is used, it inserts 10 X's into **v**. The second time, it inserts the contents of a second vector, **v2**, into **v**. This second use is the most interesting. It takes three iterator arguments. The first specifies the point at which the insertion will occur within the invoking container. The last two point to the beginning and ending of the sequence to be inserted.

Storing Class Objects in a Vector

Although the preceding examples have only stored objects of the built-in types in a vector, **vector**s are not limited to this. They can store any type of objects, including those of classes that you create. Here is an example that uses a **vector** to store objects that hold the daily temperature highs for a week. Notice that **DailyTemp** defines the default constructor and that overloaded versions of < and == are provided. Remember, depending upon how your compiler implements the STL, these (or other) comparison operators may need to be defined.

```
// Store a class object in a vector.
#include <iostream>
#include <vector>
#include <cstdlib>
using namespace std;
```

```
class DailyTemp {
  int temp;
public:
  DailyTemp() { temp = 0; }
  DailyTemp(int x) { temp = x; }

  DailyTemp &operator=(int x) {
    temp = x; return *this;
  }

  double get_temp() { return temp; }
};

bool operator<(DailyTemp a, DailyTemp b)
{
  return a.get_temp() < b.get_temp();
}

bool operator==(DailyTemp a, DailyTemp b)
{
  return a.get_temp() == b.get_temp();
}

int main()
{
  vector<DailyTemp> v;
  unsigned int i;

  for(i=0; i<7; i++)
    v.push_back(DailyTemp(60 + rand()%30));

  cout << "Fahrenheit temperatures:\n";
  for(i=0; i<v.size(); i++)
    cout << v[i].get_temp() << " ";

  cout << endl;

  // convert from Fahrenheit to Centigrade
  for(i=0; i<v.size(); i++)
    v[i] = (int)(v[i].get_temp()-32) * 5/9 ;

  cout << "Centigrade temperatures:\n";
  for(i=0; i<v.size(); i++)
```

```
    cout << v[i].get_temp() << " ";

  return 0;
}
```

Sample output from this program is shown here:

```
Fahrenheit temperatures:
71 77 64 70 89 64 78
Centigrade temperatures:
21 25 17 21 31 17 25
```

Vectors offer great power, safety, and flexibility, but they are less efficient than normal arrays. Thus, for most programming tasks, normal arrays will still be your first choice. But watch for situations in which the benefits of using a **vector** outweigh its costs.

Lists

The **list** class supports a bidirectional, linear list. Unlike a vector, which supports random access, a list can be accessed sequentially only. Since lists are bidirectional, they may be accessed front to back or back to front.

A **list** has this template specification:

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

Here, T is the type of data stored in the list. The allocator is specified by **Allocator**, which defaults to the standard allocator. It has the following constructors:

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

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

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

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

Some of the commonly used **list** member functions are shown in Table 24-3. Like vectors, elements may be put into a list by using the **push_back( )** function. You can put elements on the front of the list by using **push_front( )**. An element can also be

Member	Description
reference back( ); const_reference back( ) const;	Returns a reference to the last element in the list.
iterator begin( ); const_iterator begin( ) const;	Returns an iterator to the first element in the list.
void clear( );	Removes all elements from the list.
bool empty( ) const;	Returns true if the invoking list is empty and false otherwise.
iterator end( ); const_iterator end( ) const;	Returns an iterator to the end of the list.
iterator erase(iterator *i*);	Removes the element pointed to by *i*. Returns an iterator to the element after the one removed.
iterator erase(iterator *start*, iterator *end*);	Removes the elements in the range *start* to *end*. Returns an iterator to the element after the last element removed.
reference front( ); const_reference front( ) const;	Returns a reference to the first element in the list.
iterator insert(iterator *i*, const T &*val*);	Inserts *val* immediately before the element specified by *i*. An iterator to the element is returned.
void insert(iterator *i*, size_type *num*, const T &*val*)	Inserts *num* copies of *val* immediately before the element specified by *i*.
template <class InIter> void insert(iterator *i*, InIter *start*, InIter *end*);	Inserts the sequence defined by *start* and *end* immediately before the element specified by *i*.

Table 24-3. *Some Commonly Used* ***list*** *Member Functions*

Member	Description
void merge(list<T, Allocator> &*ob*); template <class Comp> void merge(list<T, Allocator> &*ob*, Comp *cmpfn*);	Merges the ordered list contained in *ob* with the ordered invoking list. The result is ordered. After the merge, the list contained in *ob* is empty. In the second form, a comparison function can be specified that determines when one element is less than another.
void pop_back( );	Removes the last element in the list.
void pop_front( );	Removes the first element in the list.
void push_back(const T &*val*);	Adds an element with the value specified by *val* to the end of the list.
void push_front(const T &*val*);	Adds an element with the value specified by *val* to the front of the list.
void remove(const T &*val*);	Removes elements with the value *val* from the list.
void reverse( );	Reverses the invoking list.
size_type size( ) const;	Returns the number of elements currently in the list.

void sort( );
template <class Comp>
void sort(Comp *cmpfn*);
	Sorts the list. The second form sorts the list using the comparison function *cmpfn* to determine when one element is less than another.

void splice(iterator *i*, list<T, Allocator> &*ob*);	The contents of *ob* are inserted into the invoking list at the location pointed to by *i*. After the operation, *ob* is empty.
void splice(iterator *i*, list<T, Allocator> &*ob*, iterator *el*);	The element pointed to by *el* is removed from the list *ob* and stored in the invoking list at the location pointed to by *i*.
void splice(iterator *i*, list<T, Allocator> &*ob*, iterator *start*, iterator *end*);	The range defined by *start* and *end* is removed from *ob* and stored in the invoking list beginning at the location pointed to by *i*.

Table 24-3. *Some Commonly Used* ***list*** *Member Functions* (continued)

inserted into the middle of a list by using **insert( )**. Two lists may be joined using **splice( )**. One list may be merged into another using **merge( )**.

For maximum flexibility and portability, any object that will be held in a list should define a default constructor. It should also define the < operator, and possibly other comparison operators. The precise requirements for an object that will be stored in a list vary from compiler to compiler, so you will need to check your compiler's documentation.

Here is a simple example of a **list**.

```
// List basics.
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<int> lst; // create an empty list
  int i;

  for(i=0; i<10; i++) lst.push_back(i);

  cout << "Size = " << lst.size() << endl;

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

  // change contents of list
  p = lst.begin();
  while(p != lst.end()) {
    *p = *p + 100;
    p++;
  }

  cout << "Contents modified: ";
  p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
```

```
    p++;
  }

  return 0;
}
```

The output produced by this program is shown here:

```
Size = 10
Contents: 0 1 2 3 4 5 6 7 8 9

Contents modified: 100 101 102 103 104 105 106 107 108 109
```

This program creates a list of integers. First, an empty **list** object is created. Next, 10 integers are put into the list. This is accomplished using the **push_back( )** function, which puts each new value on the end of the existing list. Next, the size of the list and the list itself is displayed. The list is displayed via an iterator, using the following code:

```
list<int>::iterator p = lst.begin();
while(p != lst.end()) {
  cout << *p << " ";
  p++;
}
```

Here, the iterator **p** is initialized to point to the start of the list. Each time through the loop, **p** is incremented, causing it to point to the next element. The loop ends when **p** points to the end of the list. This code is essentially the same as was used to cycle through a vector using an iterator. Loops like this are common in STL code, and the fact that the same constructs can be used to access different types of containers is part of the power of the STL.

Understanding end( )

Now is a good time to emphasize a somewhat unexpected attribute of the **end( )** container function. **end( )** does not return a pointer to the last element in a container. Instead, it returns a pointer *one past* the last element. Thus, the last element in a container is pointed to by **end( ) - 1**. This feature allows us to write very efficient algorithms that cycle through all of the elements of a container, including the last one, using an iterator. When the iterator has the same value as the one returned by **end( )**, we know that all elements have been accessed. However, you must keep this feature in mind since it may seem a bit counterintuitive. For example, consider the following program, which displays a list forward and backward.

```
// Understanding end().
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<int> lst; // create an empty list
  int i;

  for(i=0; i<10; i++) lst.push_back(i);

  cout << "List printed forwards:\n";
  list<int>::iterator p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }
  cout << "\n\n";

  cout << "List printed backwards:\n";
  p = lst.end();
  while(p != lst.begin()) {
    p--; // decrement pointer before using
    cout << *p << " ";
  }

  return 0;
}
```

The output produced by this program is shown here:

```
List printed forwards:
0 1 2 3 4 5 6 7 8 9

List printed backwards:
9 8 7 6 5 4 3 2 1 0
```

The code that displays the list in the forward direction is the same as we have been using. But pay special attention to the code that displays the list in reverse order. The iterator **p** is initially set to the end of the list through the use of the **end( )** function. Since **end( )** returns an iterator to an object that is one past the last object actually

stored in the list, **p** must be decremented before it is used. This is why **p** is decremented before the **cout** statement inside the loop, rather than after. Remember: **end( )** does not return a pointer to the last object in the list; it returns a pointer that is one past the last value in the list.

push_front( ) vs. push_back( )

You can build a list by adding elements to either the end or the start of the list. So far, we have been adding elements to the end by using **push_back( )**. To add elements to the start, use **push_front( )**. For example,

```
/* Demonstrating the difference between
   push_back() and push_front(). */
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<int> lst1, lst2;
  int i;

  for(i=0; i<10; i++) lst1.push_back(i);
  for(i=0; i<10; i++) lst2.push_front(i);

  list<int>::iterator p;

  cout << "Contents of lst1:\n";
  p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p << " ";
    p++;
  }
  cout << "\n\n";

  cout << "Contents of lst2:\n";
  p = lst2.begin();
  while(p != lst2.end()) {
    cout << *p << " ";
    p++;
  }
```

```
  return 0;
}
```

The output produced by this program is shown here:

```
Contents of lst1:
0 1 2 3 4 5 6 7 8 9

Contents of lst2:
9 8 7 6 5 4 3 2 1 0
```

Since **lst2** is built by putting elements onto its front, the resulting list is in the reverse order of **lst1**, which is built by putting elements onto its end.

Sort a List

A list may be sorted by calling the **sort( )** member function. The following program creates a list of random integers and then puts the list into sorted order.

```
// Sort a list.
#include <iostream>
#include <list>
#include <cstdlib>
using namespace std;

int main()
{
  list<int> lst;
  int i;

  // create a list of random integers
  for(i=0; i<10; i++)
    lst.push_back(rand());

  cout << "Original contents:\n";
  list<int>::iterator p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }
```

```
  cout << endl << endl;

  // sort the list
  lst.sort();

  cout << "Sorted contents:\n";
  p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

Here is sample output produced by the program:

```
Original contents:
41 18467 6334 26500 19169 15724 11478 29358 26962 24464

Sorted contents:
41 6334 11478 15724 18467 19169 24464 26500 26962 29358
```

Merging One List with Another

One ordered list may be merged with another. The result is an ordered list that contains the contents of the two original lists. The new list is left in the invoking list, and the second list is left empty. The next example merges two lists. The first contains the even numbers between 0 and 9. The second contains the odd numbers. These lists are then merged to produce the sequence 0 1 2 3 4 5 6 7 8 9.

```
// Merge two lists.
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<int> lst1, lst2;
  int i;
```

```
  for(i=0; i<10; i+=2) lst1.push_back(i);
  for(i=1; i<11; i+=2) lst2.push_back(i);

  cout << "Contents of lst1:\n";
  list<int>::iterator p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl << endl;

  cout << "Contents of lst2:\n";
  p = lst2.begin();
  while(p != lst2.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl << endl;

  // now, merge the two lists
  lst1.merge(lst2);
  if(lst2.empty())
    cout << "lst2 is now empty\n";

  cout << "Contents of lst1 after merge:\n";
  p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

The output produced by this program is shown here:

```
Contents of lst1:
0 2 4 6 8

Contents of lst2:
1 3 5 7 9
```

```
lst2 is now empty
Contents of lst1 after merge:
0 1 2 3 4 5 6 7 8 9
```

One other thing to notice about this example is the use of the **empty( )** function. It returns true if the invoking container is empty. Since **merge( )** removes all of the elements from the list being merged, it will be empty after the merge is completed, as the program output confirms.

Storing Class Objects in a List

Here is an example that uses a list to store objects of type **myclass**. Notice that the <, >, !=, and == are overloaded for objects of type **myclass**. (For some compilers, you will not need to define all of these. For other compilers, you may need to define additional operators.) The STL uses these functions to determine the ordering and equality of objects in a container. Even though a list is not an ordered container, it still needs a way to compare elements when searching, sorting, or merging.

```
// Store class objects in a list.
#include <iostream>
#include <list>
#include <cstring>
using namespace std;

class myclass {
  int a, b;
  int sum;
public:
  myclass() { a = b = 0; }
  myclass(int i, int j) {
    a = i;
    b = j;
    sum = a + b;
  }
  int getsum() { return sum; }

  friend bool operator<(const myclass &o1,
                        const myclass &o2);
  friend bool operator>(const myclass &o1,
                        const myclass &o2);
  friend bool operator==(const myclass &o1,
```

```
                          const myclass &o2);
  friend bool operator!=(const myclass &o1,
                          const myclass &o2);
};

bool operator<(const myclass &o1, const myclass &o2)
{
  return o1.sum < o2.sum;
}

bool operator>(const myclass &o1, const myclass &o2)
{
  return o1.sum > o2.sum;
}

bool operator==(const myclass &o1, const myclass &o2)
{
  return o1.sum == o2.sum;
}

bool operator!=(const myclass &o1, const myclass &o2)
{
  return o1.sum != o2.sum;
}

int main()
{
  int i;

  // create first list
  list<myclass> lst1;
  for(i=0; i<10; i++) lst1.push_back(myclass(i, i));

  cout << "First list: ";
  list<myclass>::iterator p = lst1.begin();
  while(p != lst1.end()) {
    cout << p->getsum() << " ";
    p++;
```

```
  }
  cout << endl;

  // create a second list
  list<myclass> lst2;
  for(i=0; i<10; i++) lst2.push_back(myclass(i*2, i*3));

  cout << "Second list: ";
  p = lst2.begin();
  while(p != lst2.end()) {
    cout << p->getsum() << " ";
    p++;
  }
  cout << endl;

  // now, merget lst1 and lst2
  lst1.merge(lst2);

  // display merged list
  cout << "Merged list: ";
  p = lst1.begin();
  while(p != lst1.end()) {
    cout << p->getsum() << " ";
    p++;
  }

  return 0;
}
```

The program creates two lists of **myclass** objects and displays the contents of each list. It then merges the two lists and displays the result. The output from this program is shown here:

```
First list: 0 2 4 6 8 10 12 14 16 18
Second list: 0 5 10 15 20 25 30 35 40 45
Merged list: 0 0 2 4 5 6 8 10 10 12 14 15 16 18 20 25 30 35 40 45
```

Maps

The **map** class supports an associative container in which unique keys are mapped with values. In essence, a key is simply a name that you give to a value. Once a value has been stored, you can retrieve it by using its key. Thus, in its most general sense, a map is a list of key/value pairs. The power of a map is that you can look up a value given its key. For example, you could define a map that uses a person's name as its key and stores that person's telephone number as its value. Associative containers are becoming more popular in programming.

As mentioned, a map can hold only unique keys. Duplicate keys are not allowed. To create a map that allows nonunique keys, use **multimap**.

The **map** container has the following template specification:

```
template <class Key, class T, class Comp = less<Key>,
          class Allocator = allocator<pair<const key, T> > class map
```

Here, **Key** is the data type of the keys, **T** is the data type of the values being stored (mapped), and **Comp** is a function that compares two keys. This defaults to the standard **less( )** utility function object. **Allocator** is the allocator (which defaults to **allocator**) .

A **map** has the following constructors:

```
explicit map(const Comp &cmpfn = Comp( ),
             const Allocator &a = Allocator( ) );
map(const map<Key, T, Comp, Allocator> &ob);
template <class InIter> map(InIter start, InIter end,
             const Comp &cmpfn = Comp( ), const Allocator &a = Allocator( ));
```

The first form constructs an empty map. The second form constructs a map that contains the same elements as *ob*. The third form constructs a map that contains the elements in the range specified by the iterators *start* and *end*. The function specified by *cmpfn*, if present, determines the ordering of the map.

In general, any object used as a key should define a default constructor and overload the < operator and any other necessary comparison operators. The specific requirements vary from compiler to compiler.

The following comparison operators are defined for **map**.

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

Several of the **map** member functions are shown in Table 24-4. In the descriptions, **key_type** is the type of the key, and **value_type** represents **pair<Key, T>**.

Member	Description
iterator begin( ); const_iterator begin( ) const;	Returns an iterator to the first element in the map.
void clear( );	Removes all elements from the map.
size_type count(const key_type &*k*) const;	Returns the number of times *k* occurs in the map (1 or zero).
bool empty( ) const;	Returns true if the invoking map is empty and false otherwise.
iterator end( ); const_iterator end( ) const;	Returns an iterator to the end of the list.
void erase(iterator *i*);	Removes the element pointed to by *i*.
void erase(iterator *start*, iterator *end*);	Removes the elements in the range *start* to *end*.
size_type erase(const key_type &*k*)	Removes from the map elements that have keys with the value *k*.
iterator find(const key_type &*k*); const_iterator find(const key_type &*k*) const;	Returns an iterator to the specified key. If the key is not found, then an iterator to the end of the map is returned.
iterator insert(iterator *i*, const value_type &*val*);	Inserts *val* at or after the element specified by *i*. An iterator to the element is returned.
template <class InIter> void insert(InIter *start*, InIter *end*)	Inserts a range of elements.
pair<iterator, bool> insert(const value_type &*val*);	Inserts *val* into the invoking map. An iterator to the element is returned. The element is inserted only if it does not already exist. If the element was inserted, pair<iterator, true> is returned. Otherwise, pair<iterator, false> is returned.

Table 24-4. *Several Commonly Used* ***map*** *Member Functions*

```
  return 0;
}
```

Notice the use of the **pair** template class to construct the key/value pairs. The data types specified by **pair** must match those of the **map** into which the pairs are being inserted.

Once the map has been initialized with keys and values, you can search for a value given its key by using the **find( )** function. **find( )** returns an iterator to the matching element or to the end of the map if the key is not found. When a match is found, the value associated with the key is contained in the **second** member of **pair**.

In the preceding example, key/value pairs were constructed explicitly, using **pair<char, int>**. While there is nothing wrong with this approach, it is often easier to use **make_pair( )**, which constructs a pair object based upon the types of the data used as parameters. For example, assuming the previous program, this line of code will also insert key/value pairs into **m**.

```
m.insert(make_pair((char)('A'+i), 65+i));
```

Here, the cast to **char** is needed to override the automatic conversion to **int** when **i** is added to 'A.' Otherwise, the type determination is automatic.

Storing Class Objects in a Map

As with all of the containers, you can use a map to store objects of types that you create. For example, the next program creates a simple phone directory. That is, it creates a map of names with their numbers. To do this, it creates two classes called **name** and **number**. Since a map maintains a sorted list of keys, the program also defines the < operator for objects of type **name**. In general, you must define the < operator for any classes that you will use as the key. (Some compilers may require that additional comparison operators be defined.)

```
// Use a map to create a phone directory.
#include <iostream>
#include <map>
#include <cstring>
using namespace std;

class name {
  char str[40];
public:
  name() { strcpy(str, ""); }
```

```
  name(char *s) { strcpy(str, s); }
  char *get() { return str; }

};

// Must define less than relative to name objects.
bool operator<(name a, name b)
{
   return strcmp(a.get(), b.get()) < 0;
}

class phoneNum {
  char str[80];
public:
  phoneNum() { strcmp(str, ""); }
  phoneNum(char *s) { strcpy(str, s); }
  char *get() { return str; }
};


int main()
{
  map<name, phoneNum> directory;

 // put names and numbers into map
  directory.insert(pair<name, phoneNum>(name("Tom"),
                   phoneNum("555-4533")));
  directory.insert(pair<name, phoneNum>(name("Chris"),
                   phoneNum("555-9678")));
  directory.insert(pair<name, phoneNum>(name("John"),
            phoneNum("555-8195")));
  directory.insert(pair<name, phoneNum>(name("Rachel"),
                   phoneNum("555-0809")));

  // given a name, find number
  char str[80];
  cout << "Enter name: ";
  cin >> str;

  map<name, phoneNum>::iterator p;

  p = directory.find(name(str));
```

```
  if(p != directory.end())
    cout << "Phone number: " <<  p->second.get();
  else
    cout << "Name not in directory.\n";

  return 0;
}
```

Here is a sample run:

```
Enter name: Rachel
Phone number: 555-0809.
```

In the program, each entry in the map is a character array that holds a null- terminated string. Later in this chapter, you will see an easier way to write this program that uses the standard **string** type.

Algorithms

As explained, algorithms act on containers. Although each container provides support for its own basic operations, the standard algorithms provide more extended or complex actions. They also allow you to work with two different types of containers at the same time. To have access to the STL algorithms, you must include **<algorithm>** in your program.

The STL defines a large number of algorithms, which are summarized in Table 24-5. All of the algorithms are template functions. This means that they can be applied to any type of container. All of the algorithms in the STL are covered in Part Four. The following sections demonstrate a representative sample.

Counting

One of the most basic operations that you can perform on a sequence is to count its contents. To do this, you can use either **count( )** or **count_if( )**. Their general forms are shown here:

```
template <class InIter, class T>
    ptrdiff_t count(InIter start, InIter end, const T &val);
template <class InIter, class UnPred>
    ptrdiff_t count_if(InIter start, InIter end, UnPred pfn);
```

The type **ptrdiff_t** is defined as some form of integer.

Algorithm	Purpose
adjacent_find	Searches for adjacent matching elements within a sequence and returns an iterator to the first match.
binary_search	Performs a binary search on an ordered sequence.
copy	Copies a sequence.
copy_backward	Same as **copy( )** except that it moves the elements from the end of the sequence first.
count	Returns the number of elements in the sequence.
count_if	Returns the number of elements in the sequence that satisfy some predicate.
equal	Determines if two ranges are the same.
equal_range	Returns a range in which an element can be inserted into a sequence without disrupting the ordering of the sequence.
fill and fill_n	Fills a range with the specified value.
find	Searches a range for a value and returns an iterator to the first occurrence of the element.
find_end	Searches a range for a subsequence. It returns an iterator to the end of the subsequence within the range.
find_first_of	Finds the first element within a sequence that matches an element within a range.
find_if	Searches a range for an element for which a user-defined unary predicate returns true.
for_each	Applies a function to a range of elements.
generate and generate_n	Assign elements in a range the values returned by a generator function.
includes	Determines if one sequence includes all of the elements in another sequence.
inplace_merge	Merges a range with another range. Both ranges must be sorted in increasing order. The resulting sequence is sorted.
iter_swap	Exchanges the values pointed to by its two iterator arguments.
lexicographical_compare	Alphabetically compares one sequence with another.

Table 24-5. *The STL Algorithms*

Algorithm	Purpose
lower_bound	Finds the first point in the sequence that is not less than a specified value.
make_heap	Constructs a heap from a sequence.
max	Returns the maximum of two values.
max_element	Returns an iterator to the maximum element within a range.
merge	Merges two ordered sequences, placing the result into a third sequence.
min	Returns the minimum of two values.
min_element	Returns an iterator to the minimum element within a range.
mismatch	Finds first mismatch between the elements in two sequences. Iterators to the two elements are returned.
next_permutation	Constructs next permutation of a sequence.
nth_element	Arranges a sequence such that all elements less than a specified element *E* come before that element and all elements greater than *E* come after it.
partial_sort	Sorts a range.
partial_sort_copy	Sorts a range and then copies as many elements as will fit into a resulting sequence.
partition	Arranges a sequence such that all elements for which a predicate returns true come before those for which the predicate returns false.
pop_heap	Exchanges the first and last –1 elements and then rebuilds the heap.
prev_permutation	Constructs previous permutation of a sequence.
push_heap	Pushes an element onto the end of a heap.
random_shuffle	Randomizes a sequence.
remove, remove_if, remove_copy, and remove_copy_if	Removes elements from a specified range.
replace, replace_copy, replace_if, and replace_copy_if	Replaces elements within a range.

Table 24-5. *The STL Algorithms* (continued)

Algorithm	Purpose
reverse and reverse_copy	Reverses the order of a range.
rotate and rotate_copy	Left-rotates the elements in a range.
search	Searches for subsequence within a sequence.
search_n	Searches for a sequence of a specified number of similar elements.
set_difference	Produces a sequence that contains the difference between two ordered sets.
set_intersection	Produces a sequence that contains the intersection of the two ordered sets.
set_symmetric_difference	Produces a sequence that contains the symmetric difference between the two ordered sets.
set_union	Produces a sequence that contains the union of the two ordered sets.
sort	Sorts a range.
sort_heap	Sorts a heap within a specified range.
stable_partition	Arranges a sequence such that all elements for which a predicate returns true come before those for which the predicate returns false. The partitioning is stable. This means that the relative ordering of the sequence is preserved.
stable_sort	Sorts a range. The sort is stable. This means that equal elements are not rearranged.
swap	Exchanges two values.
swap_ranges	Exchanges elements in a range.
transform	Applies a function to a range of elements and stores the outcome in a new sequence.
unique and unique_copy	Eliminates duplicate elements from a range.
upper_bound	Finds the last point in a sequence that is not greater than some value.

Table 24-5. *The STL Algorithms (continued)*

The **count( )** algorithm returns the number of elements in the sequence beginning at *start* and ending at *end* that match *val*. The **count_if( )** algorithm returns the number of elements in the sequence beginning at *start* and ending at *end* for which the unary predicate *pfn* returns true.

The following program demonstrates **count( )**.

```
// Demonstrate count().
#include <iostream>
#include <vector>
#include <cstdlib>
#include <algorithm>
using namespace std;

int main()
{
  vector<bool> v;
  unsigned int i;

  for(i=0; i < 10; i++) {
   if(rand() % 2) v.push_back(true);
   else v.push_back(false);
  }

  cout << "Sequence:\n";
  for(i=0; i<v.size(); i++)
    cout << boolalpha << v[i] << " ";
  cout << endl;

  i = count(v.begin(), v.end(), true);
  cout << i << " elements are true.\n";

  return 0;
}
```

This program displays the following output:

```
Sequence:
true true false false true false false false false false
3 elements are true.
```

The program begins by creating a vector comprised of randomly generated true and false values. Next, **count( )** is used to count the number of **true** values.

This next program demonstrates **count_if( )**. It creates a vector containing the numbers 1 through 19. It then counts those that are evenly divisible by 3. To do this, it creates a unary predicate called **dividesBy3( )**, which returns **true** if its argument is evenly divisible by 3.

```
// Demonstrate count_if().
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

/* This is a unary predicate that determines
   if number is divisible by 3. */
bool dividesBy3(int i)
{
  if((i%3) == 0) return true;

  return false;
}

int main()
{
  vector<int> v;
  int i;

  for(i=1; i < 20; i++) v.push_back(i);

  cout << "Sequence:\n";
  for(i=0; i<v.size(); i++)
    cout << v[i] << " ";
  cout << endl;

  i = count_if(v.begin(), v.end(), dividesBy3);
  cout << i << " numbers are divisible by 3.\n";

  return 0;
}
```

This program produces the following output.

```
Sequence:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
6 numbers are divisible by 3.
```

Notice how the unary predicate **dividesBy3( )** is coded. All unary predicates receive as a parameter an object that is of the same type as that stored in the container upon which the predicate is operating. The predicate must then return a **true** or **false** result based upon this object.

Removing and Replacing Elements

Sometimes it is useful to generate a new sequence that consists of only certain items from an original sequence. One algorithm that does this is **remove_copy( )**. Its general form is shown here:

```
template <class InIter, class OutIter, class T>
  OutIter remove_copy(InIter start, InIter end,
                      OutIter result, const T &val);
```

The **remove_copy( )** algorithm copies elements from the specified range, removing those that are equal to *val*. It puts the result into the sequence pointed to by *result* and returns an iterator to the end of the result. The output container must be large enough to hold the result.

To replace one element in a sequence with another when a copy is made, use **replace_copy( )**. Its general form is shown here:

```
template <class InIter, class OutIter, class T>
  OutIter replace_copy(InIter start, InIter end,
                       OutIter result, const T &old, const T &new);
```

The **replace_copy( )** algorithm copies elements from the specified range, replacing elements equal to *old* with *new*. It puts the result into the sequence pointed to by *result* and returns an iterator to the end of the result. The output container must be large enough to hold the result.

The following program demonstrates **remove_copy( )** and **replace_copy( )**. It creates a sequence of characters. It then removes all of the spaces from the sequence. Next, it replaces all spaces with colons.

```
// Demonstrate remove_copy and replace_copy.
#include <iostream>
#include <vector>
#include <algorithm>
```

```
using namespace std;

int main()
{
  char str[] = "The STL is power programming.";
  vector<char> v, v2(30);
  unsigned int i;

  for(i=0; str[i]; i++) v.push_back(str[i]);

  // **** demonstrate remove_copy ****
  cout << "Input sequence:\n";
  for(i=0; i<v.size(); i++) cout << v[i];
  cout << endl;

  // remove all spaces
  remove_copy(v.begin(), v.end(), v2.begin(), ' ');

  cout << "Result after removing spaces:\n";
  for(i=0; i<v2.size(); i++) cout << v2[i];
  cout << endl << endl;

  // **** now, demonstrate replace_copy ****
  cout << "Input sequence:\n";
  for(i=0; i<v.size(); i++) cout << v[i];
  cout << endl;

  // replace spaces with colons
  replace_copy(v.begin(), v.end(), v2.begin(), ' ', ':');


  cout << "Result after replacing spaces with colons:\n";
  for(i=0; i<v2.size(); i++) cout << v2[i];
  cout << endl << endl;

  return 0;
}
```

The output produced by this program is shown here.

```
Input sequence:
The STL is power programming.
Result after removing spaces:
TheSTLispowerprogramming.

Input sequence:
The STL is power programming.
Result after replacing spaces with colons:
The:STL:is:power:programming.
```

Reversing a Sequence

An often useful algorithm is **reverse( )**, which reverses a sequence. Its general form is

template <class BiIter> void reverse(BiIter *start*, BiIter *end*);

The **reverse( )** algorithm reverses the order of the range specified by *start* and *end*.

The following program demonstrates **reverse( )**.

```
// Demonstrate reverse.
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  vector<int> v;
  unsigned int i;

  for(i=0; i<10; i++) v.push_back(i);

  cout << "Initial: ";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  reverse(v.begin(), v.end());

  cout << "Reversed: ";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";

  return 0;
}
```

The output from this program is shown here:

```
Initial: 0 1 2 3 4 5 6 7 8 9
Reversed: 9 8 7 6 5 4 3 2 1 0
```

Transforming a Sequence

One of the more interesting algorithms is **transform( )** because it modifies each element in a range according to a function that you provide. The **transform( )** algorithm has these two general forms:

```
template <class InIter, class OutIter, class Func)
  OutIter transform(InIter start, InIter end, OutIter result, Func unaryfunc);
template <class InIter1, class InIter2, class OutIter, class Func)
  OutIter transform(InIter1 start1, InIter1 end1, InIter2 start2,
                    OutIter result, Func binaryfunc);
```

The **transform( )** algorithm applies a function to a range of elements and stores the outcome in *result*. In the first form, the range is specified by *start* and *end*. The function to be applied is specified *unaryfunc*. This function receives the value of an element in its parameter, and it must return its transformation. In the second form, the transformation is applied using a binary operator function that receives the value of an element from the sequence to be transformed in its first parameter and an element from the second sequence as its second parameter. Both versions return an iterator to the end of the resulting sequence.

The following program uses a simple transformation function called **reciprocal( )** to transform the contents of a list of numbers into their reciprocals. Notice that the resulting sequence is stored in the same list that provided the original sequence.

```
// An example of the transform algorithm.
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

// A simple transformation function.
double reciprocal(double i) {
  return 1.0/i; // return reciprocal
}

int main()
```

```
{

  list<double> vals;
  int i;

  // put values into list
  for(i=1; i<10; i++) vals.push_back((double)i);

  cout << "Original contents of vals:\n";
  list<double>::iterator p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  cout << endl;

  // transform vals
  p = transform(vals.begin(), vals.end(),
                vals.begin(), reciprocal);

  cout << "Transformed contents of vals:\n";
  p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

The output produced by the program is shown here:

```
Original contents of vals:
1 2 3 4 5 6 7 8 9
Transformed contents of vals:
1 0.5 0.333333 0.25 0.2 0.166667 0.142857 0.125 0.111111
```

As you can see, each element in **vals** has been transformed into its reciprocal.

Using Function Objects

As explained at the start of this chapter, the STL supports (and extensively utilizes) function objects. Recall that function objects are simply classes that define **operator( )**. The STL provides many built-in function objects, such as **less**, **minus**, etc. It also allows you to define your own function objects. Frankly, it is beyond the scope of this book to fully describe all of the issues surrounding the creation and use of function objects. Fortunately, as the preceding examples have shown, you can make significant use of the STL without ever creating a function object. However, since function objects are a main ingredient of the STL, it is important to have a general understanding.

Unary and Binary Function Objects

Just as there are unary and binary predicates, there are unary and binary function objects. A unary function object requires one argument; a binary function object requires two. You must use the type of object required. For example, if an algorithm is expecting a binary function object, you must pass it a binary function object.

Using the Built-in Function Objects

The STL provides a rich assortment of built-in function objects. The binary function objects are shown here:

plus	minus	multiplies	divides	modulus
equal_to	not_equal_to	greater	greater_equal	less
less_equal	logical_and	logical_or		

Here are the unary function objects:

logical_not	negate

The function objects perform the operations specified by their names. The only one that may not be self-evident is **negate( )**, which reverses the sign of its argument.

The built-in function objects are template classes that overload **operator( )**, which returns the result of the specified operation on whatever type of data you select. For example, to invoke the binary function object **plus( )** for **float** data, use this syntax:

```
plus<float>()
```

The built-in function objects use the header **<functional>**.

Let's begin with a simple example. The following program uses the **transform( )** algorithm (described in the preceding section) and the **negate( )** function object to reverse the sign of a list of values.

```
// Use a unary function object.
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;

int main()
{
  list<double> vals;
  int i;

  // put values into list
  for(i=1; i<10; i++) vals.push_back((double)i);

  cout << "Original contents of vals:\n";
  list<double>::iterator p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl;

  // use the negate function object
  p = transform(vals.begin(), vals.end(),
                vals.begin(),
                negate<double>()); // call function object

  cout << "Negated contents of vals:\n";
  p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

This program produces the following output:

```
Original contents of vals:
1 2 3 4 5 6 7 8 9
Negated contents of vals:
-1 -2 -3 -4 -5 -6 -7 -8 -9
```

In the program, notice how **negate( )** is invoked. Since **vals** is a list of **double** values, **negate( )** is called using **negate<double>( )**. The **transform( )** algorithm automatically calls **negate( )** for each element in the sequence. Thus, the single parameter to **negate( )** receives as its argument an element from the sequence.

The next program demonstrates the use of the binary function object **divides( )**. It creates two lists of double values and has one divide the other. This program uses the binary form of the **transform( )** algorithm.

```
// Use a binary function object.
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;

int main()
{
  list<double> vals;
  list<double> divisors;
  int i;

  // put values into list
  for(i=10; i<100; i+=10) vals.push_back((double)i);
  for(i=1; i<10; i++) divisors.push_back(3.0);

  cout << "Original contents of vals:\n";
  list<double>::iterator p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  cout << endl;

  // transform vals
```

```
  p = transform(vals.begin(), vals.end(),
                divisors.begin(), vals.begin(),
                divides<double>()); // call function object

  cout << "Divided contents of vals:\n";
  p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

The output from this program is shown here:

```
Original contents of vals:
10 20 30 40 50 60 70 80 90
Divided contents of vals:
3.33333 6.66667 10 13.3333 16.6667 20 23.3333 26.6667 30
```

In this case, the binary function object **divides( )** divides the elements from the first sequence by their corresponding elements from the second sequence. Thus, **divides( )** receives arguments in this order:

divides(*first, second*)

This order can be generalized. Whenever a binary function object is used, its arguments are ordered *first, second*.

Creating a Function Object

In addition to using the built-in function objects, you can create your own. To do so, you will simply create a class that overloads the **operator( )** function. However, for the greatest flexibility, you will want to use one of the following classes defined by the STL as a base class for your function objects.

```
template <class Argument, class Result> struct unary_function {
  typedef Argument argument_type;
  typedef Result result_type;
};
```

```
template <class Argument1, class Argument2, class Result>
struct binary_function {
  typedef Argument1 first_argument_type;
  typedef Argument2 second_argument_type;
  typedef Result result_type;
};
```

These template classes provide concrete type names for the generic data types used by the function object. Although they are technically a convenience, they are almost always used when creating function objects.

The following program demonstrates a custom function object. It converts the **reciprocal( )** function (used to demonstrate the **transform( )** algorithm earlier) into a function object.

```
// Create a reciprocal function object.
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;

// A simple function object.
class reciprocal: unary_function<double, double> {
public:
  result_type operator()(argument_type i)
  {
    return (result_type) 1.0/i; // return reciprocal
  }
};

int main()
{
  list<double> vals;
  int i;

  // put values into list
  for(i=1; i<10; i++) vals.push_back((double)i);

  cout << "Original contents of vals:\n";
  list<double>::iterator p = vals.begin();
  while(p != vals.end()) {
```

```
    cout << *p << " ";
    p++;
  }
  cout << endl;

  // use reciprocal function object
  p = transform(vals.begin(), vals.end(),
                vals.begin(),
                reciprocal()); // call function object

  cout << "Transformed contents of vals:\n";
  p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

Notice two important aspects of **reciprocal( )**. First, it inherits the base class **unary_function**. This gives it access to the **argument_type** and **result_type** types. Second, it defines **operator( )** such that it returns the reciprocal of its argument. In general, to create a function object, simply inherit the proper base class and overload **operator( )** as required. It really is that easy.

Using Binders

When using a binary function object, it is possible to bind a value to one of the arguments. This can be useful in many situations. For example, you may wish to remove all elements from a sequence that are greater than some value, such as 8. To do this, you need some way to bind 8 to the right-hand operand of the function object **greater( )**. That is, you want **greater( )** to perform the comparison

val > 8

for each element of the sequence. The STL provides a mechanism, called *binders*, that accomplishes this.

There are two binders: **bind2nd( )** and **bind1st( )**. They take these general forms:

bind1st(*binfunc_obj, value*)
bind2nd(*binfunc_obj, value*)

Here, *binfunc_obj* is a binary function object. **bind1st( )** returns a unary function object that has *binfunc_obj*'s left-hand operand bound to *value*. **bind2nd( )** returns a unary function object that has *binfunc_obj*'s right-hand operand bound to *value*. The **bind2nd( )** binder is by far the most commonly used. In either case, the outcome of a binder is a unary function object that is bound to the value specified.

To demonstrate the use of a binder, we will use the **remove_if( )** algorithm. It removes elements from a sequence based upon the outcome of a predicate. It has this prototype:

```
template <class ForIter, class UnPred>
  ForIter remove_if(ForIter start, ForIter end, UnPred func);
```

The algorithm removes elements from the sequence defined by *start* and *end* if the unary predicate defined by *func* is true. The algorithm returns a pointer to the new end of the sequence which reflects the deletion of the elements.

The following program removes all values from a sequence that are greater than the value 8. Since the predicate required by **remove_if( )** is unary, we cannot simply use the **greater( )** function object as-is because **greater( )** is a binary object. Instead, we must bind the value 8 to the second argument of **greater( )** using the **bind2nd( )** binder, as shown in the program.

```
// Demonstrate bind2nd().
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;

int main()
{
  list<int> lst;
  list<int>::iterator p, endp;

  int i;

  for(i=1; i < 20; i++) lst.push_back(i);

  cout << "Original sequence:\n";
  p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
```

```
  }
  cout << endl;

  endp = remove_if(lst.begin(), lst.end(),
                   bind2nd(greater<int>(), 8));

  cout << "Resulting sequence:\n";
  p = lst.begin();
  while(p != endp) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

The output produced by the program is shown here:

```
Original sequence:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Resulting sequence:
1 2 3 4 5 6 7 8
```

You might want to experiment with this program, trying different function objects and binding different values. As you will discover, binders expand the power of the STL in very significant ways.

One last point: There is an object related to a binder called a *negator*. The negators are **not1( )** and **not2( )**. They return the negation (i.e., the complement of) whatever predicate they modify. They have these general forms:

not1(*unary_predicate*)
not2(*binary_predicate*)

For example, if you substitute the line

```
endp = remove_if(lst.begin(), lst.end(),
                 not1(bind2nd(greater<int>(), 8)));
```

into the preceding program, it will remove all elements from **lst** that are not greater than 8.

The string Class

As you know, C++ does not support a built-in string type per se. It does, however, provide for two ways of handling strings. First, you may use the traditional, null-terminated character array with which you are already familiar. This is sometimes referred to as a *C string*. The second way is as a class object of type **string**; this is the approach examined here.

Actually, the **string** class is a specialization of a more general template class called **basic_string**. In fact, there are two specializations of **basic_string**: **string**, which supports 8-bit character strings, and **wstring**, which supports wide-character strings. Since 8-bit characters are by far the most commonly used in normal programming, **string** is the version of **basic_string** examined here.

Before looking at the **string** class, it is important to understand why it is part of the C++ library. Standard classes have not been casually added to C++. In fact, a significant amount of thought and debate has accompanied each new addition. Given that C++ already contains some support for strings as null-terminated character arrays, it may at first seem that the inclusion of the **string** class is an exception to this rule. However, this is actually far from the truth. Here is why: Null-terminated strings cannot be manipulated by any of the standard C++ operators. Nor can they take part in normal C++ expressions. For example, consider this fragment:

```
char s1[80], s2[80], s3[80];

s1 = "Alpha"; // can't do
s2 = "Beta"; // can't do
s3 = s1 + s2; // error, not allowed
```

As the comments show, in C++ it is not possible to use the assignment operator to give a character array a new value (except during initialization), nor is it possible to use the + operator to concatenate two strings. These operations must be written using library functions, as shown here:

```
strcpy(s1, "Alpha");
strcpy(s2, "Beta");
strcpy(s3, s1);
strcat(s3, s2);
```

Since null-terminated character arrays are not technically data types in their own right, the C++ operators cannot be applied to them. This makes even the most rudimentary string operations clumsy. More than anything else, it is the inability to operate on null-terminated strings using the standard C++ operators that has driven the development of a standard string class. Remember, when you define a class in C++,

you are defining a new data type that can be fully integrated into the C++ environment. This, of course, means that the operators can be overloaded relative to the new class. Therefore, by adding a standard string class, it becomes possible to manage strings in the same way as any other type of data: through the use of operators.

There is, however, one other reason for the standard string class: safety. In the hands of an inexperienced or careless programmer, it is very easy to overrun the end of an array that holds a null-terminated string. For example, consider the standard string copy function **strcpy( )**. This function contains no provision for checking the boundary of the target array. If the source array contains more characters than the target array can hold, then a program error or system crash is possible (likely). As you will see, the standard **string** class prevents such errors.

In the final analysis, there are three reasons for the inclusion of the standard **string** class: consistency (a string now defines a data type), convenience (you may use the standard C++ operators), and safety (array boundaries will not be overrun). Keep in mind that there is no reason that you should abandon normal, null-terminated strings altogether. They are still the most efficient way in which to implement strings. However, when speed is not an overriding concern, using the new **string** class gives you access to a safe and fully integrated way to manage strings.

Although not traditionally thought of as part of the STL, **string** is another container class defined by C++. This means that it supports the algorithms described in the previous section. However, strings have additional capabilities. To have access to the **string** class, you must include **<string>** in your program.

The **string** class is very large, with many constructors and member functions. Also, many member functions have multiple overloaded forms. For this reason, it is not possible to look at the entire contents of **string** in this chapter. Instead, we will examine several of its most commonly used features. Once you have a general understanding of how **string** works, you can easily explore the rest of it on your own.

The **string** class supports several constructors. The prototypes for three of its most commonly used ones are shown here:

```
string( );
string(const char *str);
string(const string &str);
```

The first form creates an empty **string** object. The second creates a **string** object from the null-terminated string pointed to by *str*. This form provides a conversion from null-terminated strings to **string** objects. The third form creates a **string** from another **string**.

A number of operators that apply to strings are defined for **string** objects, including:

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

These operators allow the use of **string** objects in normal expressions and eliminate the need for calls to functions such as **strcpy( )** or **strcat( )**, for example. In general, you can mix **string** objects with normal, null-terminated strings in expressions. For example, a **string** object can be assigned a null-terminated string.

The + operator can be used to concatenate a string object with another string object or a string object with a C-style string. That is, the following variations are supported:

```
string + string
string + C-string
C-string + string
```

The + operator can also be used to concatenate a character onto the end of a string.

The **string** class defines the constant **npos**, which is –1. This constant represents the length of the longest possible string.

The C++ string classes make string handling extraordinarily easy. For example, using **string** objects you can use the assignment operator to assign a quoted string to a **string**, the + operator to concatenate strings, and the comparison operators to compare strings. The following program illustrates these operations.

```
// A short string demonstration.
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string str1("Alpha");
  string str2("Beta");
  string str3("Omega");
  string str4;

  // assign a string
  str4 = str1;
  cout << str1 << "\n" << str3 << "\n";

  // concatenate two strings
  str4 = str1 + str2;
  cout << str4 << "\n";

  // can compare a string with a C-string
  if(str1 == "Alpha") cout << "str1 == Alpha\n";

  // compare strings
  if(str3 > str1) cout << "str3 > str1\n";
  if(str3 == str1+str2)
    cout << "str3 == str1+str2\n";

  /* A string object can also be
     assigned a normal string. */
  str1 = "This is a null-terminated string.\n";
  cout << str1;

  // create a string object using another string object
  string str5(str1);
  cout << str5;

  // input a string
  cout << "Enter a string: ";
  cin >> str5;
  cout << str5;
```

```
  return 0;
}
```

This program produces the following output:

```
Alpha
Omega
AlphaBeta
Alpha to Omega
str3 > str1
This is a null-terminated string.
This is a null-terminated string.
Enter a string: STL
STL
```

Notice the ease with which the string handling is accomplished. For example, the + is used to concatenate strings and the > is used to compare two strings. To accomplish these operations using C-style, null-terminated strings, less convenient calls to the **strcat( )** and **strcmp( )** functions would be required. Because C++ **string** objects can be freely mixed with C-style null-terminated strings, there is no disadvantage to using them in your program—and there are considerable benefits to be gained.

There is one other thing to notice in the preceding program: the size of the strings is not specified. **string** objects are automatically sized to hold the string that they are given. Thus, when assigning or concatenating strings, the target string will grow as needed to accommodate the size of the new string. It is not possible to overrun the end of the string. This dynamic aspect of **string** objects is one of the ways that they are better than standard null-terminated strings (which *are* subject to boundary overruns).

Some string Member Functions

Although most simple string operations can be accomplished using the string operators, more complex or subtle ones are accomplished using **string** member functions. While **string** has far too many member functions to discuss them all, we will examine several of the most common.

Basic String Manipulations

To assign one string to another, use the **assign( )** function. Two of its forms are shown here.

```
string &assign(const string &strob, size_type start, size_type num);
string &assign(const char *str, size_type num);
```

In the first form, *num* characters from *strob* beginning at the index specified by *start* will be assigned to the invoking object. In the second form, the first *num* characters of the null-terminated string *str* are assigned to the invoking object. In each case, a reference to the invoking object is returned. Of course, it is much easier to use the = to assign one entire string to another. You will need to use the **assign( )** function only when assigning a partial string.

You can append part of one string to another using the **append( )** member function. Two of its forms are shown here:

```
string &append(const string &strob, size_type start, size_type num);
string &append(const char *str, size_type num);
```

Here, *num* characters from *strob* beginning at the index specified by *start* will be appended to the invoking object. In the second form, the first *num* characters of the null-terminated string *str* are appended to the invoking object. In each case, a reference to the invoking object is returned. Of course, it is much easier to use the + to append one entire string to another. You will need to use the **append( )** function only when appending a partial string.

You can insert or replace characters within a string using **insert( )** and **replace( )**. The prototypes for their most common forms are shown here:

```
string &insert(size_type start, const string &strob);
string &insert(size_type start, const string &strob,
               size_type insStart, size_type num);
string &replace(size_type start, size_type num, const string &strob);
string &replace(size_type start, size_type orgNum, const string &strob,
                size_type replaceStart, size_type replaceNum);
```

The first form of **insert( )** inserts *strob* into the invoking string at the index specified by *start*. The second form of **insert( )** function inserts *num* characters from *strob* beginning at *insStart* into the invoking string at the index specified by *start*.

Beginning at *start*, the first form of **replace( )** replaces *num* characters from the invoking string, with *strob*. The second form replaces *orgNum* characters, beginning at *start*, in the invoking string with the *replaceNum* characters from the string specified by *strob* beginning at *replaceStart*. In both cases, a reference to the invoking object is returned.

You can remove characters from a string using **erase( )**. One of its forms is shown here:

string &erase(size_type *start* = 0, size_type *num* = npos);

It removes *num* characters from the invoking string beginning at *start*. A reference to the invoking string is returned.

The following program demonstrates the **insert( )**, **erase( )**, and **replace( )** functions.

```
// Demonstrate insert(), erase(), and replace().
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string str1("String handling C++ style.");
  string str2("STL Power");

  cout << "Initial strings:\n";
  cout << "str1: " << str1 << endl;
  cout << "str2: " << str2 << "\n\n";

  // demonstrate insert()
  cout << "Insert str2 into str1:\n";
  str1.insert(6, str2);
  cout << str1 << "\n\n";

  // demonstrate erase()
  cout << "Remove 9 characters from str1:\n";
  str1.erase(6, 9);
  cout << str1 <<"\n\n";

  // demonstrate replace
  cout << "Replace 8 characters in str1 with str2:\n";
  str1.replace(7, 8, str2);
  cout << str1 << endl;

  return 0;
}
```

The output produced by this program is shown here.

```
Initial strings:
str1: String handling C++ style.
str2: STL Power

Insert str2 into str1:
StringSTL Power handling C++ style.

Remove 9 characters from str1:
String handling C++ style.

Replace 8 characters in str1 with str2:
String STL Power C++ style.
```

Searching a String

The **string** class provides several member functions that search a string, including **find( )** and **rfind( )**. Here are the prototypes for the most common versions of these functions:

size_type find(const string &*strob*, size_type *start*=0) const;
size_type rfind(const string &*strob*, size_type *start*=npos) const;

Beginning at *start*, **find( )** searches the invoking string for the first occurrence of the string contained in *strob*. If found, **find( )** returns the index at which the match occurs within the invoking string. If no match is found, then **npos** is returned. **rfind( )** is the opposite of **find( )**. Beginning at *start*, it searches the invoking string in the reverse direction for the first occurrence of the string contained in *strob* (i.e, it finds the last occurrence of *strob* within the invoking string). If found, **rfind( )** returns the index at which the match occurs within the invoking string. If no match is found, **npos** is returned.

Here is a short example that uses **find( )** and **rfind( )**.

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

int main()
{
  int i;
  string s1 =
    "Quick of Mind, Strong of Body, Pure of Heart";
  string s2;
```

```
  i = s1.find("Quick");
  if(i!=string::npos) {
    cout << "Match found at " << i << endl;
    cout << "Remaining string is:\n";
    s2.assign(s1, i, s1.size());
    cout << s2;
  }
  cout << "\n\n";

  i = s1.find("Strong");
  if(i!=string::npos) {
    cout << "Match found at " << i << endl;
    cout << "Remaining string is:\n";
    s2.assign(s1, i, s1.size());
    cout << s2;
  }
  cout << "\n\n";

  i = s1.find("Pure");
  if(i!=string::npos) {
    cout << "Match found at " << i << endl;
    cout << "Remaining string is:\n";
    s2.assign(s1, i, s1.size());
    cout << s2;
  }
  cout << "\n\n";

  // find list "of"
  i = s1.rfind("of");
  if(i!=string::npos) {
    cout << "Match found at " << i << endl;
    cout << "Remaining string is:\n";
    s2.assign(s1, i, s1.size());
    cout << s2;
  }

  return 0;
}
```

The output produced by this program is shown here.

```
Match found at 0
Remaining string is:
Quick of Mind, Strong of Body, Pure of Heart

Match found at 15
Remaining string is:
Strong of Body, Pure of Heart

Match found at 31
Remaining string is:
Pure of Heart

Match found at 36
Remaining string is:
of Heart
```

Comparing Strings

To compare the entire contents of one string object to another, you will normally use the overloaded relational operators described earlier. However, if you want to compare a portion of one string to another, you will need to use the **compare( )** member function, shown here:

int compare(size_type *start*, size_type *num*, const string &*strob*) const;

Here, *num* characters in *strob*, beginning at *start*, will be compared against the invoking string. If the invoking string is less than *strob*, **compare( )** will return less than zero. If the invoking string is greater than *strob*, it will return greater than zero. If *strob* is equal to the invoking string, **compare( )** will return zero.

Obtaining a Null-Terminated String

Although **string** objects are useful in their own right, there will be times when you will need to obtain a null-terminated character-array version of the string. For example, you might use a **string** object to construct a filename. However, when opening a file, you will need to specify a pointer to a standard, null-terminated string. To solve this problem, the member function **c_str( )** is provided. Its prototype is shown here:

const char *c_str( ) const;

This function returns a pointer to a null-terminated version of the string contained in the invoking **string** object. The null-terminated string must not be altered. It is also not guaranteed to be valid after any other operations have taken place on the **string** object.

Strings Are Containers

The **string** class meets all of the basic requirements necessary to be a container. Thus, it supports the common container functions, such as **begin( )**, **end( )**, and **size( )**. It also supports iterators. Therefore, a **string** object can also be manipulated by the STL algorithms. Here is a simple example:

```
// Strings as containers.
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
  string str1("Strings handling is easy in C++");
  string::iterator p;
  unsigned int i;

  // use size()
  for(i=0; i<str1.size(); i++)
    cout << str1[i];
  cout << endl;

  // use iterator
  p = str1.begin();
  while(p != str1.end())
    cout << *p++;
  cout << endl;

  // use the count() algorithm
  i = count(str1.begin(), str1.end(), 'i');
  cout << "There are " << i << " i's in str1\n";

  // use transform() to upper case the string
  transform(str1.begin(), str1.end(), str1.begin(),
            toupper);
  p = str1.begin();
  while(p != str1.end())
    cout << *p++;
  cout << endl;
```

```
  return 0;
}
```

Output from the program is shown here:

```
Strings handling is easy in C++
Strings handling is easy in C++
There are 4 i's in str1
STRINGS HANDLING IS EASY IN C++
```

Putting Strings into Other Containers

Even though **string** is a container, objects of type string are commonly held in other STL containers, such as maps or lists. For example, here is a better way to write the telephone directory program shown earlier. It uses a map of **string** objects, rather than null-terminated strings, to hold the names and telephone numbers.

```
// Use a map of strings to create a phone directory.
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
  map<string, string> directory;

  directory.insert(pair<string, string>("Tom", "555-4533"));
  directory.insert(pair<string, string>("Chris", "555-9678"));
  directory.insert(pair<string, string>("John", "555-8195"));
  directory.insert(pair<string, string>("Rachel", "555-0809"));

  string s;
  cout << "Enter name: ";
  cin >> s;

  map<string, string>::iterator p;

  p = directory.find(s);
  if(p != directory.end())
    cout << "Phone number: " << p->second;
```

```
    else
      cout << "Name not in directory.\n";

    return 0;
}
```

Final Thoughts on the STL

The STL is an important, integral part of the C++ language. Many programming tasks can (and will) be framed in terms of it. The STL combines power with flexibility, and while its syntax is a bit complex, its ease of use is remarkable. No C++ programmer can afford to neglect the STL because it will play an important role in the way future programs are written.