Rogue Wave Standard C++ Library User's Guide, Tutorial, and Class Reference

Rogue Wave Software
Corvallis, Oregon USA

Rogue Wave Standard C++ Library User's Guide and Tutorial

for

Rogue Wave's implementation of the Standard C++ Library.

Based on ANSI's Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++.

User's Guide and Tutorial Author: Timothy A. Budd

Class Reference Authors: Wendi Minne, Tom Pearson, and Randy Smithey

Product Team:

Development:	Anna Dahan, Philippe Le Mouel, Randy Smithey
Quality Engineering:	Kevin Djang, Randall Robinson
Manuals:	Elaine Cull, Wendi Minne, Julie Prince, Randy Smithey
Support:	North Krimsley

Significant contributions by: Joe Delaney

Part # RW81-01-100096

Printing Date: October, 1996

Rogue Wave Software, Inc., 850 SW 35th St., Corvallis, Oregon, 97333 USA

Product Information: (541) 754-3010

(800) 487-3217

Technical Support: (541) 754-2311

FAX: (541) 757-6650

World Wide Web: http://www.roguewave.com

Table of Contents

Section 1. Introduction

1.1 What is the Standard C++ Library?

The International Standards Organization (ISO) and the American National Standards Institute (ANSI) are completing the process of standardizing the C++ programming language. A major result of this standardization process is the "Standard C++ Library," a large and comprehensive collection of classes and functions. This product is *Rogue Wave's* implementation of the ANSI/ISO Standard Library.

The ANSI/ISO Standard C++ Library includes the following parts:

- A large set of data structures and algorithms formerly known as the Standard Template Library (STL).
- An IOStream facility.
- A locale facility.
- A templatized ***string*** class.
- A templatized class for representing complex numbers.
- A uniform framework for describing the execution environment, through the use of a template class named ***numeric_limits*** and specializations for each fundamental data type.
- Memory management features.
- Language support features.
- Exception handling features.
- A ***valarray*** class optimized for handling numeric arrays

1.2 Does the Standard C++ Library Differ From Other Libraries?

A major portion of the Standard C++ Library is a collection of class definitions for standard data structures and a collection of algorithms commonly used to manipulate such structures. This part of the library was formerly known as the Standard Template Library or STL. The organization and design of the STL differs in almost all respects from the design of most other C++ libraries, *because it avoids encapsulation and uses almost no inheritance.*

An emphasis on encapsulation is a key hallmark of object-oriented programming. The emphasis on combining data and functionality into an object is a powerful organizational principle in software development; indeed it is *the* primary organizational technique. Through the proper use of

encapsulation, even exceedingly complex software systems can be divided into manageable units and assigned to various members of a team of programmers for development.

Inheritance is a powerful technique for permitting code sharing and software reuse, but it is most applicable when two or more classes share a common set of basic features. For example, in a graphical user interface, two types of windows may inherit from a common base window class, and the individual subclasses will provide any required unique features. In another use of inheritance, object-oriented container classes may ensure common behavior and support code reuse by inheriting from a more general class, and factoring out common member functions.

The designers of the STL decided against using an entirely object-oriented approach, and separated the tasks to be performed using common data structures from the representation of the structures themselves. This is why the STL is properly viewed as a collection of algorithms and, separate from these, a collection of data structures that can be manipulated using the algorithms.

1.3 What are the Effects of Non-Object-Oriented Design?

The STL portion of the Standard C++ Library was purposely designed with an architecture that is not object-oriented. This design has side effects, some advantageous, and some not, that developers must be aware of as they investigate how to most effectively use the library. We'll discuss a few of them here.

- **Smaller Source Code**

  There are approximately fifty different algorithms in the STL, and about a dozen major data structures. This separation has the effect of reducing the size of source code, and decreasing some of the risk that similar activities will have dissimilar interfaces. Were it not for this separation, for example, each of the algorithms would have to be re-implemented in each of the different data structures, requiring several hundred more member functions than are found in the present scheme.

- **Flexibility**

  One advantage of the separation of algorithms from data structures is that such algorithms can be used with conventional C++ pointers and arrays. Because C++ arrays are not objects, algorithms encapsulated within a class hierarchy seldom have this ability.

- **Efficiency**

  The STL in particular, and the Standard C++ Library in general, provide a low-level, "nuts and bolts" approach to developing C++ applications.

This low-level approach can be useful when specific programs require an emphasis on efficient coding and speed of execution.

- **Iterators: Mismatches and Invalidations**

  The Standard C++ Library data structures use pointer-like objects called iterators to describe the contents of a container. (These are described in detail in Section 2.) Given the library's architecture, it is not possible to verify that these iterator elements are matched; i.e., that they are derived from the same container. Using (either intentionally or by accident) a beginning iterator from one container with an ending iterator from another is a recipe for certain disaster.

  It is very important to know that iterators can become invalidated as a result of a subsequent insertion or deletion from the underlying container class. This invalidation is not checked, and use of an invalid iterator can produce unexpected results.

  Familiarity with the Standard C++ Library will help reduce the number of errors related to iterators.

- **Templates: Errors and "Code Bloat"**

  The flexibility and power of templatized algorithms are, with most compilers, purchased at a loss of precision in diagnostics. Errors in the parameter lists to generic algorithms will sometimes show up only as obscure compiler errors for internal functions that are defined many levels deep in template expansions. Again, familiarity with the algorithms and their requirements is a key to successful use of the standard library.

  Because of its heavy reliance on templates, the STL can cause programs to grow larger than expected. You can minimize this problem by learning to recognize the cost of instantiating a particular template class, and by making appropriate design decisions. Be aware that as compilers become more and more fluent in templates, this will become less of a problem.

- **Multithreading Problems**

  The Standard C++ Library must be used carefully in a multithreaded environment. Iterators, because they exist independently of the containers they operate on, cannot be safely passed between threads. Since iterators can be used to modify a non `const` container, there is no way to protect such a container if it spawns iterators in multiple threads. Use "thread-safe" wrappers, such as those provided by *Tools.h++*, if you need to access a container from multiple threads.

1.4 How Should I Use the Standard C++ Library?

Within a few years the Standard C++ Library will be the standard set of classes and libraries delivered with all ANSI-conforming C++ compilers. We have noted that the design of a large portion of the Standard C++ Library is in many ways not object-oriented. On the other hand, C++ excels as a language for manipulating objects. How do we integrate the Standard Library's non-object-oriented architecture with C++'s strengths as a language for manipulating objects?

The key is to use the right tool for each task. Object-oriented design methods and programming techniques are almost without peer as guideposts in the development of large, complex software. For the large majority of programming tasks, object-oriented techniques will remain the preferred approach. And products such as Rogue Wave's *Tools.h++ 7.0*, which encapsulates the Standard C++ Library with a familiar object-oriented interface, will provide you with the power of the Library and the advantages of object-orientation.

Use Standard C++ Library components directly when you need flexibility and/or highly efficient code. Use the more traditional approaches to object-oriented design, such as encapsulation and inheritance, when you need to model larger problem domains, and knit all the pieces into a full solution. When you need to devise an architecture for your application, *always* consider the use of encapsulation and inheritance to compartmentalize the problem. But if you discover that you need an efficient data structure or algorithm for a compact problem, such as data stream manipulation in drivers (the kind of problem that often resolves to a single class), look to the Standard C++ Library. The Standard C++ Library excels in the creation of reusable classes, where low-level constructs are needed, while traditional OOP techniques really shine when those classes are combined to solve a larger problem.

In the future, most libraries will use the Standard C++ Library as their foundation. By using the Standard C++ Library, either directly or through an encapsulation such as *Tools.h++ 7.0*, you help ensure interoperability. This is especially important in large projects that may rely on communication between several libraries. A good rule of thumb is to use the highest encapsulation level available to you, but make sure that the Standard C++ Library is available as the base for interlibrary communication and operation.

The C++ language supports a wide range of programming approaches because the problems we need to solve require that range. The language, and now the Standard C++ library that supports it, are designed to give you the power to approach each unique problem from the best possible angle. The Standard C++ Library, when combined with more traditional OOP techniques, puts a very flexible tool into the hands of anyone building a collection of C++ classes, whether those classes are intended to stand alone as a library or are tailored to a specific task.

1.5 Reading This Manual

This manual is an introduction to the Rogue Wave implementation of the Standard C++ Library. It assumes that you are already familiar with the basics features of the C++ programming language. If you are new to C++ you may wish to examine an introductory text, such as the book *The C++ Programming Language*, by Bjarne Stroustrup (Addison-Wesley, 1991).

There is a classic "chicken-and-egg" problem associated with the container class portion of the standard library. The heart of the container class library is the definition of the containers themselves, but you can't really appreciate the utility of these structures without an understanding of the algorithms that so greatly extend their functionality. On the other hand, you can't really understand the algorithms without some appreciation of the containers.

Ideally, after reading sections 2, 3 and 4 carefully, sections 5 through 11 should be read simultaneously with sections 12 and 13. Since that's not possible, simply skim over sections 5 through 11 and sections 12 and 13 to gain a superficial understanding of the overall structure, then go back and read these sections again in more detail.

1.6 Conventions

We use distinctive fonts for ***class_names*** and `function_names()` When we wish to refer to a function name or algorithm name but not draw attention to the arguments, we will follow the function name with an empty pair of parentheses. We do this even when the actual function invocation requires additional arguments. We have used the term *algorithm* to refer to the functions in the generic algorithms portion of the standard library, so as to avoid confusion with member functions, argument functions, and functions defined by the programmer. Note that both class names and function names in the standard library follow the convention of using an underline character as a separator. Throughout the text, examples and file names are printed in the same `courier font` used for function names.

In the text, it is common to omit printing the class name in the distinctive font after it has been introduced. This is intended to make the appearance of the text less visually disruptive. However, we return to the distinctive font to make a distinction between several different possibilities, as for example between the classes ***vector*** and ***list*** used as containers in constructing a ***stack***.

1.7 Using the Standard Library

Because the Standard C++ Library consists largely of template declarations, on most platforms it is only necessary to include in your programs the appropriate header files. These header files will be noted in the text that describes how to use each algorithm or class.

1.8 Running the Tutorial Programs

All the tutorial programs described in this text have been gathered together and are available as part of the distribution package. You can compile and run these programs, and use them as models for your own programming problems. Many of these example programs have been extended with additional output commands that are not reproduced here in the text. The expected output from each program is also included as part of the distribution.

Section 2. Iterators

2.1 Introduction to Iterators

Fundamental to the use of the container classes and the associated algorithms provided by the standard library is the concept of an *iterator*. Abstractly, an iterator is simply a pointer-like object used to cycle through all the elements stored in a container. Because different algorithms need to traverse containers in a variety of fashions, there are different forms of iterator. Each container class in the standard library can generate an iterator with functionality appropriate to the storage technique used in implementing the container. It is the category of iterators required as arguments that chiefly distinguishes which algorithms in the standard library can be used with which container classes.

Iterators

Iterators *are pointer-like objects, used to cycle through the elements stored in a container.*

Just as pointers can be used in a variety of ways in traditional programming, iterators are also used for a number of different purposes. An iterator can be used to denote a specific value, just as a pointer can be used to reference a specific memory location. On the other hand, a *pair* of iterators can be used to describe a *range* of values, just as two pointers can be used to describe a contiguous region of memory. In the case of iterators, however, the values being described are not necessarily physically in sequence, but are rather logically in sequence, because they are derived from the same container, and the second follows the first in the order in which the elements are maintained by the container.

Range

A range *is a sequence of values held in a container. The range is described by a pair of iterators, which define the beginning and end of the sequence.*

Conventional pointers can sometimes be *null*, meaning they point at nothing. Iterators, as well, can fail to denote any specific value. Just as it is a logical error to dereference a null pointer, it is an error to dereference an iterator that is not denoting a value.

When two pointers that describe a region in memory are used in a C++ program, it is conventional that the ending pointer is *not* considered to be part of the region. For example, an array named `x` of length ten is sometimes described as extending from `x` to `x+10`, even though the element at `x+10` is not part of the array. Instead, the pointer value `x+10` is the *past-the-end* value – the element that is the next value *after* the end of the range being described. Iterators are used similarly to describe a range. The second value is not considered to be part of the range being denoted. Instead, the second value is a *past-the-end* element, describing the next value in sequence after the final value of the range. Sometimes, as with pointers to memory, this will be an actual value in the container. Other times it may be a special value, specifically constructed for the purpose. In either case, it is not proper to dereference an iterator that is being used to specify the end of a range.

Just as with conventional pointers, the fundamental operation used to modify an iterator is the increment operator (operator `++`). When the increment operator is applied to an iterator that denotes the final value in a sequence, it will be changed to the “past the end” value. An iterator `j` is said

to be *reachable* from an iterator `i` if, after a finite sequence of applications of the expression `++i`, the iterator `i` becomes equal to `j`.

Iterator Ranges

When iterators are used to describe a range of values in a container, it is assumed (but not verified) that the second iterator is reachable from the first. Errors will occur if this is not true.

Ranges can be used to describe the entire contents of a container, by constructing an iterator to the initial element and a special "ending" iterator. Ranges can also be used to describe sub-sequences within a single container, by employing two iterators to specific values. Whenever two iterators are used to describe a range it is assumed, but not verified, that the second iterator is reachable from the first. Errors can occur if this expectation is not satisfied.

In the remainder of this section we will describe the different forms of iterators used by the standard library, as well as various other iterator-related functions.

2.2 Varieties of Iterators

There are five basic forms of iterators used in the standard library:

input iterator	read only, forward moving
output iterator	write only, forward moving
forward iterator	both read and write, forward moving
bidirectional iterator	read and write, forward and backward moving
random access iterator	read and write, random access

Iterator categories are hierarchical. Forward iterators can be used wherever input or output iterators are required, bidirectional iterators can be used in place of forward iterators, and random access iterators can be used in situations requiring bidirectionality.

A second characteristic of iterators is whether or not they can be used to modify the values held by their associated container. A *constant iterator* is one that can be used for access only, and cannot be used for modification. Output iterators are never constant, and input iterators always are. Other iterators may or may not be constant, depending upon how they are created. There are both constant and non-constant bidirectional iterators, both constant and non-constant random access iterators, and so on.

The following table summarizes specific ways that various categories of iterators are generated by the containers in the standard library.

Iterator Form	*Produced By*
input iterator	`istream_iterator`

Iterator Form	*Produced By*

output iterator	`ostream_iterator`
	`inserter`
	`front_inserter`
	`back_inserter`

bidirectional iterator	***list*** ***set*** and ***multiset*** ***map*** and ***multimap***
random access iterator	ordinary pointers ***vector*** ***deque***

In the following sections we will describe the capabilities and construction of each form of iterator.

2.2.1 Input Iterators

Input iterators are the simplest form of iterator. To understand their capabilities, consider an example program. The `find()` generic algorithm (to be described in more detail in Section 13.3.1), performs a simple linear search, looking for a specific value being held within a container. The contents of the container are described using two iterators, here called `first` and `last`. While `first` is not equal to `last` the element denoted by `first` is compared to the test value. If equal, the iterator, which now denotes the located element, is returned. If not equal, the `first` iterator is incremented, and the loop cycles once more. If the entire region of memory is examined without finding the desired value, then the algorithm returns the end-of-range iterator.

```
template <class InputIterator, class T>
InputIterator
   find (InputIterator first, InputIterator last, const T& value)
{
   while (first != last && *first != value)
      ++first;
   return first;
}
```

This algorithm illustrates three requirements for an input iterator:

- An iterator can be compared for equality to another iterator. They are equal when they point to the same position, and are otherwise not equal.
- An iterator can be dereferenced using the `*` operator, to obtain the value being denoted by the iterator.

- An iterator can be incremented, so that it refers to the next element in sequence, using the operator ++.

Notice that these characteristics can all be provided with new meanings in a C++ program, since the behavior of the given functions can all be modified by overloading the appropriate operators. Because of this overloading, iterators are possible. There are three main varieties of input iterators:

Ordinary pointers. Ordinary pointers can be used as input iterators. In fact, since we can subscript and add to ordinary pointers, they are random access values, and thus can be used either as input or output iterators. The end-of-range pointer describes the end of a contiguous region of memory, and the deference and increment operators have their conventional meanings. For example, the following searches for the value 7 in an array of integers:

```
int data[100];
   ...
int * where = find(data, data+100, 7);
```

Ordinary Pointers as Iterators

Because ordinary pointers have the same functionality as random access iterators, most of the generic algorithms in the standard library can be used with conventional C++ arrays, as well as with the containers provided by the standard library.

Note that constant pointers, pointers which do not permit the underlying array to be modified, can be created by simply placing the keyword const in a declaration.

```
const int * first = data;
const int * last = data + 100;
   // can't modify location returned by the following
const int * where = find(first, last, 7);
```

Container iterators. All of the iterators constructed for the various containers provided by the standard library are at *least* as general as input iterators. The iterator for the first element in a collection is always constructed by the member function begin(), while the iterator that denotes the "past-the-end" location is generated by the member function end(). For example, the following searches for the value 7 in a list of integers:

```
list<int>::iterator where = find(aList.begin(), aList.end(), 7);
```

Each container that supports iterators provides a type within the class declaration with the name iterator. Using this, iterators can uniformly be declared in the fashion shown. If the container being accessed is constant, or if the description const_iterator is used, then the iterator is a constant iterator.

Input stream iterators. The standard library provides a mechanism to operate on an input *stream* using an input iterator. This ability is provided by the class istream_iterator, and will be described in more detail in Section 2.3.1.

2.2.2 Output Iterators

An output iterator has the opposite function from an input iterator. Output iterators can be used to assign values in a sequence, but cannot be used to

access values. For example, we can use an output iterator in a generic algorithm that copies values from one sequence into another:

```
template <class InputIterator, class OutputIterator>
OutputIterator copy
   (InputIterator first, InputIterator last, OutputIterator result)
{
    while (first != last)
      *result++ = *first++;
    return result;
}
```

Two ranges are being manipulated here; the range of source values specified by a pair of input iterators, and the destination range. The latter, however, is specified by only a single argument. It is assumed that the destination is large enough to include all values, and errors will ensue if this is not the case.

Parallel Sequences

A number of the generic algorithms manipulate two parallel sequences. Frequently the second sequence is described using only a beginning iterator, rather than an iterator pair. It is assumed, but not checked, that the second sequence has at least as many elements as the first.

As illustrated by this algorithm, an output iterator can modify the element to which it points, by being used as the target for an assignment. Output iterators can use the dereference operator only in this fashion, and cannot be used to return or access the elements they denote.

As we noted earlier, ordinary pointers, as well as all the iterators constructed by containers in the standard library, can be used as examples of output iterators. (Ordinary pointers are random access iterators, which are a superset of output iterators.) So, for example, the following code fragment copies elements from an ordinary C-style array into a standard library vector:

```
int data[100];
vector<int> newdata(100);
   ...
copy (data, data+100, newdata.begin());
```

Just as the `istream_iterator` provided a way to operate on an input stream using the input iterator mechanism, the standard library provides a data type, `ostream_iterator`, that permits values to be written to an output stream in an iterator-like fashion. These will be described in Section 2.3.2.

Yet another form of output iterator is an *insert iterator*. An insert iterator changes the output iterator operations of dereferencing / assignment and increment into insertions into a container. This permits operations such as `copy()` to be used with variable length containers, such as lists and sets. Insert iterators will be described in more detail in Section 2.4.

2.2.3 Forward Iterators

A forward iterator combines the features of an input iterator and an output iterator. It permits values to both be accessed and modified. One function that uses forward iterators is the `replace()` generic algorithm, which replaces occurrences of specific values with other values. This algorithm is written as follows:

```
template <class ForwardIterator, class T>
```

```
void
   replace (ForwardIterator first, ForwardIterator last,
            const T& old_value, const T& new_value)
{
    while (first != last)
    {
    if (*first == old_value)
        *first = new_value;
        ++first;
    }
}
```

Ordinary pointers, as well as any of the iterators produced by containers in the standard library, can be used as forward iterators. The following, for example, replaces instances of the value 7 with the value 11 in a vector of integers.

```
    replace (aVec.begin(), aVec.end(), 7, 11);
```

2.2.4 Bidirectional Iterators

A bidirectional iterator is similar to a forward iterator, except that bidirectional iterators support the decrement operator (operator `--`), permitting movement in either a forward or a backward direction through the elements of a container. For example, we can use bidirectional iterators in a function that reverses the values of a container, placing the results into a new container.

```
template <class BidirectionalIterator, class OutputIterator>
OutputIterator
   reverse_copy (BidirectionalIterator first,
                 BidirectionalIterator last,
                 OutputIterator result)
{
    while (first != last)
    *result++ = *--last;
    return result;
}
```

As always, the value initially denoted by the `last` argument is not considered to be part of the collection.

The `reverse_copy()` function could be used, for example, to reverse the values of a linked list, and place the result into a vector:

```
   list<int> aList;
    ....
   vector<int> aVec (aList.size());
   reverse_copy (aList.begin(), aList.end(), aVec.begin() );
```

2.2.5 Random Access Iterators

Some algorithms require more functionality than the ability to access values in either a forward or backward direction. Random access iterators permit values to be accessed by subscript, subtracted one from another (to yield the

number of elements between their respective values) or modified by arithmetic operations, all in a manner similar to conventional pointers.

When using conventional pointers, arithmetic operations can be related to the underlying memory; that is, `x+10` is the memory ten elements after the beginning of `x`. With iterators the logical meaning is preserved (`x+10` is the tenth element after `x`), however the physical addresses being described may be different.

Algorithms that use random access iterators include generic operations such as sorting and binary search. For example, the following algorithm randomly shuffles the elements of a container. This is similar to, although simpler than, the function `random_shuffle()` provided by the standard library.

```
template <class RandomAccessIterator>
void
   mixup (RandomAccessIterator first, RandomAccessIterator last)
{
   while (first < last)
   {
      iter_swap(first, first + randomInteger(last - first));
      ++first;
   }
}
```

The program will cycle as long as `first` is denoting a position that occurs earlier in the sequence than the one denoted by `last`. Only random access iterators can be compared using relational operators; all other iterators can be compared only for equality or inequality. On each cycle through the loop, the expression `last - first` yields the number of elements between the two limits. The function `randomInteger()` is assumed to generate a random number between 0 and the argument. Using the standard random number generator, this function could be written as follows:

randomInteger()

The function `randomInteger` described here is used in a number of the example programs presented in later sections.

```
unsigned int randomInteger (unsigned int n)
   // return random integer greater than
   // or equal to 0 and less than n
{
   return rand() % n;
}
```

This random value is added to the iterator `first`, resulting in an iterator to a randomly selected value in the container. This value is then swapped with the element denoted by the iterator `first`.

2.2.6 Reverse Iterators

An iterator naturally imposes an order on an underlying container of values. For a ***vector*** or a ***map*** the order is given by increasing index values. For a ***set*** it is the increasing order of the elements held in the container. For a ***list*** the order is explicitly derived from the way values are inserted.

A *reverse iterator* will yield values in exactly the reverse order of those given by the standard iterators. That is, for a vector or a list, a reverse iterator will generate the last element first, and the first element last. For a set it will generate the largest element first, and the smallest element last. Strictly speaking, reverse iterators are not themselves a new category of iterator. Rather, there are reverse bidirectional iterators, reverse random access iterators, and so on.

The ***list***, ***set*** and ***map*** data types provide a pair of member functions that produce reverse bidirectional iterators. The functions `rbegin()` and `rend()` generate iterators that cycle through the underlying container in reverse order. Increments to such iterators move backward, and decrements move forward through the sequence.

Similarly, the ***vector*** and ***deque*** data types provide functions (also named `rbegin()` and `rend()`) that produce reverse random access iterators. Subscript and addition operators, as well as increments to such iterators move backward within the sequence.

2.3 Stream Iterators

Stream iterators are used to access an existing input or output stream using iterator operations.

2.3.1 Input Stream Iterators

Stream Iterators

An input stream iterator permits an input stream to be read using iterator operations. An output stream iterator similarly writes to an output stream when iterator operations are executed.

As we noted in the discussion of input iterators, the standard library provides a mechanism to turn an input stream into an input iterator. This ability is provided by the class `istream_iterator`. When declared, the four template arguments are the element type, the stream character type, the character traits type, and a type that measures the distance between elements. The latter two default to `char_traits<charT >` and `ptrdiff_t`. This is almost always the appropriate behavior. The single argument provided to the constructor for an `istream_iterator` is the stream to be accessed. Each time the `++` operator is invoked on an input stream iterator a new value from the stream is read (using the `>>` operator) and stored. This value is then available through the use of the dereference operator (operator `*`). The value constructed by `istream_iterator` when no arguments are provided to the constructor can be used as an ending iterator value. The following, for example, finds the first value 7 in a file of integer values.

```
istream_iterator<int, char> intstream(cin), eof;
istream_iterator<int, char>::iterator where =
         find(intstream, eof, 7);
```

The element denoted by an iterator for an input stream is valid only until the next element in the stream is requested. Also, since an input stream iterator is an input iterator, elements can only be accessed, they cannot be modified by assignment. Finally, elements can be accessed only once, and only in a

forward moving direction. If you want to read the contents of a stream more than one time, you must create a separate iterator for each pass.

2.3.2 Output Stream Iterators

The output stream iterator mechanism is analogous to the input stream iterator. Each time a value is assigned to the iterator, it will be written on the associated output stream, using the >> operator. To create an output stream iterator you must specify, as an argument with the constructor, the associated output stream. Values written to the output stream must recognize the stream >> operation. An optional second argument to the constructor is a string that will be used as a separator between each pair of values. The following, for example, copies all the values from a vector into the standard output, and separates each value by a space:

```
copy (newdata.begin(), newdata.end(),
      ostream_iterator<int,char> (cout, " "));
```

Simple file transformation algorithms can be created by combining input and output stream iterators and the various algorithms provided by the standard library. The following short program reads a file of integers from the standard input, removes all occurrences of the value 7, and copies the remainder to the standard output, separating each value by a new line:

```
void main()
{
   istream_iterator<int, char> input (cin), eof;
   ostream_iterator<int, char> output (cout, "\n");

   remove_copy (input, eof, output, 7);
}
```

2.4 Insert Iterators

Assignment to the dereferenced value of an output iterator is normally used to *overwrite* the contents of an existing location. For example, the following invocation of the function copy() transfers values from one vector to another, although the space for the second vector was already set aside (and even initialized) by the declaration statement:

```
vector<int> a(10);
vector<int> b(10);
   ...
copy (a.begin(), a.end(), b.begin());
```

Even structures such as lists can be overwritten in this fashion. The following assumes that the list named c has at least ten elements. The initial ten locations in the list will be replaced by the contents of the vector a.

```
list<int> c;
   ...
copy (a.begin(), a.end(), c.begin());
```

With structures such as lists and sets, which are dynamically enlarged as new elements are added, it is frequently more appropriate to *insert* new values into the structure, rather than to *overwrite* existing locations. A type of adaptor called an *insert iterator* allows us to use algorithms such as `copy()` to insert into the associated container, rather than overwrite elements in the container. The output operations of the iterator are changed into insertions into the associated container. The following, for example, inserts the values of the vector `a` into an initially empty list:

```
list<int> d;

copy (a.begin(), a.end(), front_inserter(d));
```

There are three forms of insert iterators, all of which can be used to change a *copy* operation into an *insert* operation. The iterator generated using `front_inserter`, shown above, inserts values into the front of the container. The iterator generated by `back_inserter` places elements into the back of the container. Both forms can be used with ***list***s and ***deque***s, but not with ***set***s or ***map***s. `back_inserter`, but not `front_inserter`, can be used with ***vector***.

The third, and most general form, is `inserter`, which takes two arguments; a container and an iterator within the container. This form copies elements into the specified location in the container. (For a list, this means elements are copied immediately before the specified location). This form can be used with all the structures for which the previous two forms work, as well as with sets and maps.

The following simple program illustrates the use of all three forms of insert iterators. First, the values 3, 2 and 1 are inserted into the front of an initially empty list. Note that as it is inserted, each value becomes the new front, so that the resultant list is ordered 1, 2, 3. Next, the values 7, 8 and 9 are inserted into the end of the list. Finally, the `find()` operation is used to locate an iterator that denotes the 7 value, and the numbers 4, 5 and 6 are inserted immediately prior. The result is the list of numbers from 1 to 9 in order.

```
void main() {
   int threeToOne [ ] = {3, 2, 1};
   int fourToSix [ ] = {4, 5, 6};
   int sevenToNine [ ] = {7, 8, 9};

   list<int> aList;

                          // first insert into the front
                          // note that each value becomes new front
   copy (threeToOne, threeToOne+3, front_inserter(aList));

                          // then insert into the back
   copy (sevenToNine, sevenToNine+3, back_inserter(aList));

                          // find the seven, and insert into middle
   list<int>::iterator seven = find(aList.begin(), aList.end(), 7);
   copy (fourToSix, fourToSix+3, inserter(aList, seven));

                          // copy result to output
   copy (aList.begin(), aList.end(),
```

```
        ostream_iterator<int,char>(cout, " "));
   cout << endl;
}
```

Observe that there is an important and subtle difference between the iterators created by `inserter(aList, aList.begin())` and `front_inserter(aList).` The call on `inserter(aList, aList.begin())` copies values in sequence, adding each one to the front of a list, whereas `front_inserter(aList)` copies values making each value the new front. The result is that `front_inserter(aList)` reverses the order of the original sequence, while `inserter(aList, aList.begin())` retains the original order.

2.5 Iterator Operations

The standard library provides two functions that can be used to manipulate iterators. The function `advance()` takes an iterator and a numeric value as argument, and modifies the iterator by moving the given amount.

```
void advance (InputIterator & iter, Distance & n);
```

For random access iterators this is the same as `iter + n;` however, the function is useful because it is designed to operate with all forms of iterator. For forward iterators the numeric distance must be positive, whereas for bidirectional or random access iterators the value can be either positive or negative. The operation is efficient (constant time) only for random access iterators. In all other cases it is implemented as a loop that invokes either the operators `++` or `--` on the iterator, and therefore takes time proportional to the distance traveled. The `advance()` function does not check to ensure the validity of the operations on the underlying iterator.

The second function, `distance(),` returns the number of iterator operations necessary to move from one element in a sequence to another. The description of this function is as follows:

```
void distance (InputIterator first, InputIterator last,
               Distance &n);
```

The result is returned in the third argument, which is passed by reference. Distance will *increment* this value by the number of times the operator `++` must be executed to move from `first` to `last`. Always be sure that the variable passed through this argument is properly initialized before invoking the function.

Section 3.

Functions and Predicates

3.1 Functions

A number of algorithms provided in the standard library require functions as arguments. A simple example is the algorithm `for_each(),` which invokes a function, passed as an argument, on each value held in a container. The following, for example, applies the `printElement()` function to produce output describing each element in a list of integer values:

```
void printElement (int value)
{
   cout << "The list contains " << value << endl;
}

main ()
{
   list<int> aList;
      ...
   for_each (aList.begin(), aList.end(), printElement);
}
```

Binary functions take two arguments, and are often applied to values from two different sequences. For example, suppose we have a list of strings and a list of integers. For each element in the first list we wish to replicate the string the number of times given by the corresponding value in the second list. We could perform this easily using the function `transform()` from the standard library. First, we define a binary function with the desired characteristics:

```
string stringRepeat (const string & base, int number)
                    // replicate base the given number of times
{
   string result;   // initially the result is empty
   while (number--)  result += base;
   return result;
}
```

The following call on `transform()` then produces the desired effect:

```
list<string> words;
list<int> counts;
   ...
transform (words.begin(), words.end(),
   counts.begin(), words.begin(), stringRepeat);
```

Transforming the words `one, two, three` with the values 3, 2, 3 would yield the result `oneoneone, twotwo, threethreethree`.

3.2 Predicates

A *predicate* is simply a function that returns either a boolean (true/false) value or an integer value. Following the normal C convention, an integer value is assumed to be true if non-zero, and false otherwise. An example

function might be the following, which takes as argument an integer and returns `true` if the number represents a leap year, and `false` otherwise:

```
bool isLeapYear (unsigned int year)
                    // return true if year is leap year
{
                    // millennia are leap years
   if (0 == year % 1000) return true;
                   // every fourth century is
   if (0 == year % 400) return true;
                   // every fourth year is
   if (0 == year % 4) return true;
                   // otherwise not
   return false;
}
```

A predicate is used as an argument, for example, in the generic algorithm named `find_if()`. This algorithm returns the first value that satisfies the predicate, returning the end-of-range value if no such element is found. Using this algorithm, the following locates the first leap year in a list of years:

```
list<int>::iterator firstLeap =
   find_if(aList.begin(), aList.end(), isLeapYear);
```

3.3 Function Objects

A *function object* is an instance of a class that defines the parenthesis operator as a member function. There are a number of situations where it is convenient to substitute function objects in place of functions. When a function object is used as a function, the parenthesis operator is invoked whenever the function is called.

To illustrate, consider the following class definition:

```
class biggerThanThree
 {
   public:
   bool operator () (int val)
   { return val > 3; }
};
```

If we create an instance of class ***biggerThanThree***, every time we reference this object using the function call syntax, the parenthesis operator member function will be invoked. The next step is to generalize this class, by adding a constructor and a constant data field, which is set by the constructor.

```
class biggerThan {
   public:
      const int testValue;
      biggerThan (int x) : testValue(x) { }

      bool operator () (int val)
      { return val > testValue; }
};
```

The result is a general “bigger than X” function, where the value of X is determined when we create an instance of the class. We can do so, for

example, as an argument to one of the generic functions that require a predicate. In this manner the following will find the first value in a list that is larger than 12:

```
list<int>::iterator firstBig =
   find_if (aList.begin(), aList.end(), biggerThan(12));
```

Three of the most common reasons to use function objects in place of ordinary functions are to employ an existing function object provided by the standard library instead of a new function, to improve execution by using inline function calls, or to allow a function object to access or set state information that is held by an object. We will give examples of each.

The following table illustrates the function objects provided by the standard library.

Name	Implemented operations
arithmetic functions	

`plus`
`minus`
`multiplies`
`divides`
`modulus`
`negate`
	addition `x + y`
	subtraction `x - y`
	multiplication `x * y`
	division `x / y`
	remainder `x % y`
	negation `- x`

comparison functions	

`equal_to`
`not_equal_to`
`greater`
`less`
`greater_equal`
`less_equal`
	equality test `x == y`
	inequality test `x != y`
	greater comparison `x > y`
	less-than comparison `x < y`
	greater than or equal comparison `x >= y`
	less than or equal comparison `x <= y`

logical functions	

`logical_and`
`logical_or`
`logical_not`
	logical conjunction `x && y`
	logical disjunction `x || y`
	logical negation `! x`

Let's look at a couple of examples that show how these might be used. The first example uses `plus()` to compute the by-element addition of two lists of integer values, placing the result back into the first list. This can be performed by the following:

```
transform (listOne.begin(), listOne.end(), listTwo.begin(),
   listOne.begin(), plus<int>() );
```

The second example negates every element in a vector of boolean values:

```
transform (aVec.begin(), aVec.end(), aVec.begin(),
   logical_not<bool>() );
```

The base classes used by the standard library in the definition of the functions shown in the preceding table are available for the creation of new unary and binary function objects. These base classes are defined as follows:

Location of the Class Definitions

The class definitions for unary_function and binary_function can be incorporated by #including `functional`.

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

template <class Arg1, class Arg2, class Result>
struct binary_function {
   typedef Arg1 first_argument_type;
   typedef Arg2 second_argument_type;
   typedef Result result_type;
};
```

An example of the use of these functions is found in Section 6.3. Here we want to take a binary function of type “Widget” and an argument of type integer, and compare the widget identification number against the integer value. A function to do this is written in the following manner:

```
struct WidgetTester : binary_function<Widget, int, bool> {
public:
   bool operator () (const Widget & wid, int testid) const
      { return wid.id == testid; }
};
```

A second reason to consider using function objects instead of functions is faster code. In many cases an invocation of a function object, such as the examples given in the calls on `transform()` presented earlier, can be expanded in-line, eliminating the overhead of a function call.

The third major reason to use a function object in place of a function is when each invocation of the function must remember some state set by earlier invocations. An example of this occurs in the creation of a generator, to be used with the generic algorithm `generate()`. A *generator* is simply a function that returns a different value each time it is invoked. The most commonly used form of generator is a random number generator, but there are other uses for the concept. A sequence generator simply returns the values of an increasing sequence of natural numbers (1, 2, 3, 4 and so on). We can call this object ***iotaGen*** after the similar operation in the programming language APL, and define it as follows:

```
class iotaGen {
public:
   iotaGen (int start = 0) : current(start) { }
   int operator () () { return current++; }
private:
   int current;
};
```

An iota object maintains a current value, which can be set by the constructor, or defaults to zero. Each time the function-call operator is invoked, the current value is returned, and also incremented. Using this object, the following call on the standard library function `generate()` will initialize a vector of 20 elements with the values 1 through 20:

```
vector<int> aVec(20);
generate (aVec.begin(), aVec.end(), iotaGen(1));
```

Using Function Objects to Store References

A more complex illustration of the use of a function object occurs in the radix sorting example program given as an illustration of the use of the list data type in Section 6.3. In this program references are initialized in the function object, so that during the sequence of invocations the function object can access and modify local values in the calling program.

3.4 Function Adaptors

A *function adaptor* is an instance of a class that adapts a global or member function so that the function can be used as a function object (a function adaptor may also be used to alter the behavior of a function or function object, as is the case in the next section). Each function adaptor provides a constructor that takes a global or member function. The adaptor also provides a parenthesis operator that forwards its call to that associated global or member function.

The pointer_to_unary_functoin and pointer_to_binary_function templates adapt global functions of one or two arguments. These adaptors can be applied directly, or you can use the ptr_fun function template to construct the appropriate adaptor automatically. For instance, I can adapt a simple times3 function and apply it to a vector of integers as follows:

```
int times3(int x) {
  return 3*x;
}

int a{} {1,2,3,4,5};
vector<int> v(a,a+5), v2;

transform(v.begin(),v.end(),v2.end(),ptr_fun(times3));
```

Alternatively, I could have applied the adaptor, and then passed the new, adapted, function object to my vector.

```
pointer_to_unary_function<int,int> pf(times3);
transform(v.begin(),v.end(),v2.end(),pf);
```

Here you can see the advantage of allowing the compiler to deduce the types needed by pointer_to_unary_function through the use of ptr_fun.

The ***mem_fun*** family of templates adapts member functions, rather than global functions. For instance, if I have a set of lists, and I want to sort each list in the set, I can use `mem_fun_t` (or more simply `mem_fun`) to apply the list sort member function to each element in the set.

```
set<list<int>* > s;

// Initialize the set with lists
…
// Sort each list in the set.
for_each(s.begin(),s.end(),mem_fun(&list<int>::sort));

// Now each list in the set is sorted
```

This is necessary because the generic sort algorithm cannot be used on a list. This is also the simplest way to access any polymorphic characteristics of an object held in a standard container. For instance I might invoke a virtual draw function on a collection of objects that are all part of the canonical 'shape' hierarchy like this:

```
// shape hierarchy
class shape {
   virtual void draw();
};

class circle : public shape {
   void draw();
};

class square : public shape {
  void draw();
};

//  Assemble a vector of shapes
circle c;
square s;
vector<shape*> v;
v.push_back(&s);
v.push_back(&c);

// Call draw on each one
for_each(v.begin(),v.end(), mem_fun(&shape::draw));
```

Similarly to the global function adaptors, each member function adaptor consists of a class template and an associated function template. The class is the actual adaptor, while the function simplifies the use of the class by constructing instances of that class on the fly. For instance, in the above

example I could have constructed a mem_fun_t myself, and then passed that to the for_each algorithm:

```
mem_fun_t<shape> mf(&shape::draw);
for_each(v.begin(),v.end(),mf);
```

Again, you can see that the ***mem_fun*** function template simplifies the use of the mem_fun_t adaptor by allowing the compiler to deduce the type needed by mem_fun_t.

The library provides member function adaptors for functions with zero arguments (as above) and one argument. This can be easily extended to member functions with more arguments.

3.5 Negators and Binders

Negators and binders are function adaptors that are used to build new function objects out of existing function objects. Almost always, these are applied to functions as part of the process of building an argument list prior to invoking yet another function or generic algorithm.

The negators not1() and not2() take a unary and a binary predicate function object, respectively, and create a new function object that will yield the complement of the original. For example, using the widget tester function object defined in the previous section, the function object

```
not2(WidgetTester())
```

yields a binary predicate which takes exactly the same arguments as the widget tester, and which is true when the corresponding widget tester would be false, and false otherwise. Negators work only with function objects defined as subclasses of the classes unary_function and binary_function, given earlier.

A Hot Idea

The idea described here by the term binder is in other contexts often described by the term curry. *This is not, as some people think, because it is a hot idea. Instead, it is named after the computer scientist Haskell P. Curry, who used the concept extensively in an influential book on the theory of computation in the 1930's. Curry himself attributed the idea to Moses Schönfinkel, leaving one to wonder why we don't instead refer to binders as "Schönfinkels."*

A binder takes a two-argument function, and binds either the first or second argument to a specific value, thereby yielding a one-argument function. The underlying function must be a subclass of class `binary_function`. The binder `bind1st()` binds the first argument, while the binder `bind2nd()` binds the second.

For example, the binder `bind2nd(greater<int>(), 5)` creates a function object that tests for being larger than 5. This could be used in the following, which yields an iterator representing the first value in a list larger than 5:

```
list<int>::iterator where = find_if(aList.begin(), aList.end(),
          bind2nd(greater<int>(), 5));
```

Combining a binder and a negator, we can create a function that is true if the argument is divisible by 3, and false otherwise. This can be used to remove all the multiples of 3 from a list.

```
list<int>::iterator where = remove_if (aList.begin(), aList.end(),
          not1(bind2nd(modulus<int>(), 3)));
```

A binder is used to tie the widget number of a call to the binary function `WidgetTester(),` yielding a one-argument function that takes only a widget as argument. This is used to find the first widget that matches the given widget type:

```
list<Widget>::iterator wehave =
   find_if(on_hand.begin(), on_hand.end(),
      bind2nd(WidgetTester(), wid));
```

Section 4.

Container Classes

4.1 Overview

The standard library provides no fewer than ten alternative forms of container. In this section we will briefly describe the varieties, considering the characteristics of each, and discuss how you might go about selecting which container to use in solving a particular problem. Subsequent sections will then go over each of the different containers in more detail.

The following chart shows the ten container types provided by the standard library, and gives a short description of the most significant characteristic for each.

Name	Characteristic
`vector`	random access to elements, efficient insertions at end
`list`	efficient insertion and removal throughout
`deque`	random access, efficient insertion at front or back
`set`	elements maintained in order, efficient test for inclusion, insertion and removal
`multiset`	set with repeated copies
`map`	access to values via keys, efficient insertion and removal
`multimap`	map permitting duplicate keys
`stack`	insertions and removals only from top
`queue`	insertion at back, removal from front
`priority queue`	efficient access and removal of largest value

4.2 Selecting a Container

The following series of questions can help you determine which type of container is best suited for solving a particular problem.

How are values going to be accessed?

If random access is important, than a ***vector*** or a ***deque*** should be used. If sequential access is sufficient, then one of the other structures may be suitable.

Is the order in which values are maintained in the collection important?

There are a number of different ways values can be sequenced. If a strict ordering is important throughout the life of the container, then the ***set*** data structure is an obvious choice, as insertions into a set are automatically placed in order. On the other hand, if this ordering is important only at one point (for example, at the end of a long series of insertions), then it might be easier to place the values into a ***list*** or ***vector***, then sort the resulting structure at the appropriate time. If the order that values are held in the

structure is related to the order of insertion, then a ***stack***, ***queue***, or ***list*** may be the best choice.

Will the size of the structure vary widely over the course of execution?

If true, then a ***list*** or ***set*** might be the best choice. A ***vector*** or ***deque*** will continue to maintain a large buffer even after elements have been removed from the collection. Conversely, if the size of the collection remains relatively fixed, than a vector or deque will use less memory than will a list or set holding the same number of elements.

Is it possible to estimate the size of the collection?

The ***vector*** data structure provides a way to pre-allocate a block of memory of a given size (using the `reserve()` member function). This ability is not provided by the other containers.

Is testing to see whether a value is contained in the collection a frequent operation?

If so, then the ***set*** or ***map*** containers would be a good choice. Testing to see whether a value is contained in a set or map can be performed in a very small number of steps (logarithmic in the size of the container), whereas testing to see if a value is contained in one of the other types of collections might require comparing the value against every element being stored by the container.

Is the collection indexed? That is, can the collection be viewed as a series of key/value pairs?

If the keys are integers between 0 and some upper limit, a ***vector*** or ***deque*** should be employed. If, on the other hand, the key values are some other ordered data type (such as characters, strings, or a user-defined type), the ***map*** container can be used.

Can values be related to each other?

All values stored in any container provided by the standard library must be able to test for equality against another similar value, but not all need to recognize the relational less-than operator. However, if values cannot be ordered using the relational less-than operator, they cannot be stored in a ***set*** or a ***map***.

Is finding and removing the largest value from the collection a frequent operation?

If the answer is “yes,” the ***priority queue*** is the best data structure to use.

At what positions are values inserted into or removed from the structure?

If values are inserted into or removed from the middle, then a ***list*** is the best choice. If values are inserted only at the beginning, a ***deque*** or a ***list*** is the preferred choice. If values are inserted or removed only at the end, a ***stack*** or ***queue*** may be a logical choice.

Is a frequent operation the merging of two or more sequences into one?

If so, a ***set*** or a ***list*** would seem to be the best choice, depending whether the collection is maintained in order. Merging two sets is a very efficient operation. If the collections are not ordered, but the efficient `splice()` member function from class list can be used, then the list data type is to be preferred, since this operation is not provided in the other containers.

In many situations any number of different containers may be applicable to a given problem. In such cases one possibility is to compare actual execution timings to determine which alternative is best.

4.3 Memory Management Issues

Containers in the standard library can maintain a variety of different types of elements. These include the fundamental data types (`integer`, `char`, and so on), pointers, or user-defined types. Containers cannot hold references. In general, memory management is handled automatically by the standard container classes, with little interaction by the programmer.

Values are placed into a container using the copy constructor. For most container classes, the element type held by the container must also define a default constructor. Generic algorithms that copy into a container (such as `copy()`) use the assignment operator.

When an entire container is duplicated (for example, through invoking a copy constructor or as the result of an assignment), every value is copied into the new structure using (depending on the structure) either the assignment operator or a copy constructor. Whether such a result is a “deep copy” or a “shallow copy,” it is controlled by the programmer, who can provide the assignment operator with whatever meaning is desired. Memory for structures used internally by the various container classes is allocated and released automatically and efficiently.

If a destructor is defined for the element type, this destructor will be invoked when values are removed from a container. When an entire collection is destroyed, the destructor will be invoked for each remaining value being held by the container.

A few words should be said about containers that hold pointer values. Such collections are not uncommon. For example, a collection of pointers is the only way to store values that can potentially represent either instances of a class or instances of a subclass. Such a collection is encountered in an example problem discussed in Section 11.3.

In these cases the container is responsible only for maintaining the pointer values themselves. It is the responsibility of the programmer to manage the memory for the values being referenced by the pointers. This includes making certain the memory values are properly allocated (usually by invoking the `new` operator), that they are not released while the container

holds references to them, and that they are properly released once they have been removed from the container.

4.4 Container Types Not Found in the Standard Library

There are a number of "classic" container types that are not found in the standard library. In most cases, the reason is that the containers that have been provided can easily be adapted to a wide variety of uses, including those traditionally solved by these alternative collections.

There is no *tree* collection that is described as such. However, the ***set*** data type is internally implemented using a form of binary search tree. For most problems that would be solved using trees, the set data type is an adequate substitute.

The set data type is specifically ordered, and there is no provision for performing set operations (union, intersection, and so on) on a collection of values that cannot be ordered (for example, a set of complex numbers). In such cases a ***list*** can be used as a substitute, although it is still necessary to write special set operation functions, as the generic algorithms cannot be used in this case.

There are no *multidimensional arrays*. However, vectors can hold other vectors as elements, so such structures can be easily constructed.

There are no *graphs*. However, one representation for graphs can be easily constructed as a map that holds other maps. This type of structure is described in the sample problem discussed in Section 9.3.2.

There are no *sparse arrays*. A novel solution to this problem is to use the graph representation discussed in Section 9.3.2.

There are no *hash tables*. A hash table provides amortized constant time access, insertion and removal of elements, by converting access and removal operations into indexing operations. However, hash tables can be easily constructed as a vector (or deque) that holds lists (or even sets) as elements. A similar structure is described in the radix sort sample problem discussed in Section 7.3, although this example does not include invoking the hash function to convert a value into an index.

In short, while not providing every conceivable container type, the containers in the standard library represent those used in the solution of most problems, and a solid foundation from which further structures can be constructed.

Section 5.

vector and vector<bool>

5.1 The vector Data Abstraction

The ***vector*** container class generalizes the concept of an ordinary C array. Like an array, a vector is an indexed data structure, with index values that range from 0 to one less than the number of elements contained in the structure. Also like an array, values are most commonly assigned to and extracted from the vector using the `subscript` operator. However, the vector differs from an array in the following important respects:

- A vector has more "self-knowledge" than an ordinary array. In particular, a vector can be queried about its size, about the number of elements it can potentially hold (which may be different from its current size), and so on.
- The size of the vector can change dynamically. New elements can be inserted on to the end of a vector, or into the middle. Storage management is handled efficiently and automatically. It is important to note, however, that while these abilities are provided, insertion into the middle of a vector is not as efficient as insertion into the middle of a ***list*** (Section 6). If many insertion operations are to be performed, the ***list*** container should be used instead of the ***vector*** data type.

The ***vector*** container class in the standard library should be compared and contrasted to the ***deque*** container class we will describe in more detail in Section 7. Like a vector, a deque (pronounced "deck") is an indexed data structure. The major difference between the two is that a deque provides efficient insertion at either the beginning or the end of the container, while a vector provides efficient insertion only at the end. In many situations, either structure can be used. Use of a vector generally results in a smaller executable file, while, depending upon the particular set of operations being performed, use of a deque may result in a slightly faster program.

5.1.1 Include Files

Whenever you use a ***vector***, you must include the `vector` header file.

```
# include <vector>
```

5.2 Vector Operations

Each of the member functions provided by the vector data type will shortly be described in more detail. Note that while member functions provide basic operations, the utility of the data structure is greatly extended through the use of the generic algorithms described in Sections 12 and 14.

5.2.1 Declaration and Initialization of Vectors

Requirements of an Element Type

Elements that are held by a vector must define a default constructor (constructor with no arguments), as well as a copy constructor. Although not used by functions in the vector class, some of the generic algorithms also require vector elements to recognize either the equivalence operator (operator ==) or the relational less-than operator (operator <).

Because it is a template class, the declaration of a vector must include a designation of the component type. This can be a primitive language type (such as integer or double), a pointer type, or a user-defined type. In the latter case, the user-defined type *must* implement a default constructor, as this constructor is used to initialize newly created elements. A copy constructor, either explicitly or implicitly defined, must also exist for the container element type. Like an array, a vector is most commonly declared with an integer argument that describes the number of elements the vector will hold:

```
vector<int> vec_one(10);
```

The constructor used to create the vector in this situation is declared as `explicit`, which prevents it being used as a conversion operator. (This is generally a good idea, since otherwise an integer might unintentionally be converted into a vector in certain situations.)

There are a variety of other forms of constructor that can also be used to create vectors. In addition to a size, the constructor can provide a constant value that will be used to initialize each new vector location. If no size is provided, the vector initially contains no elements, and increases in size automatically as elements are added. The copy constructor creates a clone of a vector from another vector.

```
vector<int> vec_two(5, 3);      // copy constructor
vector<int> vec_three;
vector<int> vec_four(vec_two);  //  initialization by assignment
```

A vector can also be initialized using elements from another collection, by means of a beginning and ending iterator pair. The arguments can be any form of iterator; thus collections can be initialized with values drawn from any of the container classes in the standard library that support iterators.

```
vector <int> vec_five (aList.begin(), aList.end());
```

A vector can be assigned the values of another vector, in which case the target receives a copy of the argument vector.

```
vec_three = vec_five;
```

The `assign()` member function is similar to an assignment, but is more versatile and, in some cases, requires more arguments. Like an assignment, the existing values in the container are deleted, and replaced with the values specified by the arguments. There are two forms of `assign()`. The first takes two iterator arguments that specify a sub-sequence of an existing container. The values from this sub-sequence then become the new elements in the receiver. The second version of `assign()` takes a count and an optional value of the container element type. After the call the container will hold only the number of elements specified by the count, which are equal to either the default value for the container type or the initial value specified.

```
vec_six.assign(list_ten.begin(), list_ten.end());
vec_four.assign(3, 7);      // three copies of the value 7
vec_five.assign(12);        // twelve copies of value zero
```

If a destructor is defined for the container element type, the destructor will be called for each value removed from the collection.

Finally, two vectors can exchange their entire contents by means of the `swap()` operation. The argument container will take on the values of the receiver, while the receiver will assume those of the argument. A swap is very efficient, and should be used, where appropriate, in preference to an explicit element-by-element transfer.

```
vec_three.swap(vec_four);
```

Constructors and Iterators

Because it requires the ability to define a method with a template argument different from the class template, some compilers may not yet support the initialization of containers using iterators. In the mean time, while compiler technology catches up with the standard library definition, the Rogue Wave version of the standard library will support conventional pointers and vector iterators in this manner.

5.2.2 Type Definitions

The class ***vector*** includes a number of type definitions. These are most commonly used in declaration statements. For example, an iterator for a vector of integers can be declared in the following fashion:

```
vector<int>::iterator location;
```

In addition to `iterator`, the following types are defined:

`value_type`	The type associated with the elements the vector maintains.
`const_iterator`	An iterator that does not allow modification of the underlying sequence.
`reverse_iterator`	An iterator that moves in a backward direction.
`const_reverse_iterator`	A combination constant and reverse iterator.
`reference`	A reference to an underlying element.

const_reference	A reference to an underlying element that will not permit the element to be modified.
size_type	An unsigned integer type, used to refer to the size of containers.
difference_type	A signed integer type, used to describe distances between iterators.
allocator_type	The type of allocator used to manage memory for the vector.

5.2.3 Subscripting a Vector

The value being maintained by a vector at a specific index can be accessed or modified using the subscript operator, just like an ordinary array. And, like arrays, there currently are no attempts to verify the validity of the index values (although this may change in future releases). Indexing a constant vector yields a constant reference. Attempts to index a vector outside the range of legal values will generate unpredictable and spurious results:

```
cout << vec_five[1] << endl;
vec_five[1] = 17;
```

The member function at() can be used in place of the subscript operator. It takes exactly the same arguments as the subscript operator, and returns exactly the same values.

The member function front() returns the first element in the vector, while the member function back() yields the last. Both also return constant references when applied to constant vectors.

```
cout << vec_five.front() << " ... " << vec_five.back() << endl;
```

5.2.4 Extent and Size-Changing Operations

There are, in general, three different "sizes" associated with any vector. The first is the number of elements currently being held by the vector. The second is the maximum size to which the vector can be expanded without requiring that new storage be allocated. The third is the upper limit on the size of any vector. These three values are yielded by the member functions size(), capacity(), and max_size(), respectively.

```
cout << "size: " << vec_five.size() << endl;
cout << "capacity: " << vec_five.capacity() << endl;
cout << "max_size: " << vec_five.max_size() << endl;
```

The maximum size is usually limited only by the amount of available memory, or the largest value that can be described by the data type size_type. The current size and capacity are more difficult to characterize. As we will note in the next section, elements can be added to or removed from a vector in a variety of ways. When elements are removed from a

vector, the memory for the vector is generally not reallocated, and thus the size is decreased but the capacity remains the same. A subsequent insertion does not force a reallocation of new memory if the original capacity is not exceeded.

An insertion that causes the size to exceed the capacity generally results in a new block of memory being allocated to hold the vector elements. Values are then copied into this new memory using the assignment operator appropriate to the element type, and the old memory is deleted. Because this can be a potentially costly operation, the ***vector*** data type provides a means for the programmer to specify a value for the capacity of a vector. The member function `reserve()` is a directive to the vector, indicating that the vector is expected to grow to at least the given size. If the argument used with `reserve()` is larger than the current capacity, then a reallocation occurs and the argument value becomes the new capacity. (It may subsequently grow even larger; the value given as the argument need not be a bound, just a guess.) If the capacity is already in excess of the argument, then no reallocation takes place. Invoking `reserve()` does not change the size of the vector, nor the element values themselves (with the exception that they may potentially be moved should reallocation take place).

```
vec_five.reserve(20);
```

A reallocation invalidates all references, pointers, and iterators referring to elements being held by a vector.

The member function `empty()` returns true if the vector currently has a size of zero (regardless of the capacity of the vector). Using this function is generally more efficient than comparing the result returned by `size()` to zero.

```
cout << "empty is " << vec_five.empty() << endl;
```

The member function `resize()` changes the size of the vector to the value specified by the argument. Values are either added to or erased from the end of the collection as necessary. An optional second argument can be used to provide the initial value for any new elements added to the collection. If a destructor is defined for the element type, the destructor will be called for any values that are removed from the collection.

```
            // become size 12, adding values of 17 if necessary
vec_five.resize (12, 17);
```

Memory Management

A vector stores values in a single large block of memory. A deque, on the other hand, employs a number of smaller blocks. This difference may be important on machines that limit the size of any single block of memory, because in such cases a deque will be able to hold much larger collections than are possible with a vector.

5.2.5 Inserting and Removing Elements

As we noted earlier, the class ***vector*** differs from an ordinary array in that a vector can, in certain circumstances, increase or decrease in size. When an insertion causes the number of elements being held in a vector to exceed the capacity of the current block of memory being used to hold the values, then a new block is allocated and the elements are copied to the new storage.

Costly Insertions

Even adding a single element to a vector can, in the worst case, require time proportional to the number of elements in the vector, as each element is moved to a new location. If insertions are a prominent feature of your current problem, then you should explore the possibility of using containers, such as lists or sets, which are optimized for insert operations.

A new element can be added to the back of a vector using the function `push_back()`. If there is space in the current allocation, this operation is very efficient (constant time).

```
vec_five.push_back(21);   // add element 21 to end of collection
```

The corresponding removal operation is `pop_back()`, which decreases the size of the vector, but does not change its capacity. If the container type defines a destructor, the destructor will be called on the value being eliminated. Again, this operation is very efficient. (The class ***deque*** permits values to be added and removed from both the back and the front of the collection. These functions are described in Section 7, which discusses deques in more detail.)

More general insertion operations can be performed using the `insert()` member function. The location of the insertion is described by an iterator; insertion takes place immediately preceding the location denoted. A fixed number of constant elements can be inserted by a single function call. It is much more efficient to insert a block of elements in a single call, than to perform a sequence of individual insertions, because with a single call at most one allocation will be performed.

```
                                  // find the location of the 7
vector<int>::iterator where =
      find(vec_five.begin(), vec_five.end(), 7);
                             // then insert the 12 before the 7
vec_five.insert(where, 12);
vec_five.insert(where, 6, 14); // insert six copies of 14
```

The most general form of the `insert()` member function takes a position and a pair of iterators that denote a sub-sequence from another container. The range of values described by the sequence is inserted into the vector. Again, because at most a single allocation is performed, using this function is preferable to using a sequence of individual insertions.

```
vec_five.insert (where, vec_three.begin(), vec_three.end());
```

In addition to the pop_back() member function, which removes elements from the end of a vector, a function exists that removes elements from the middle of a vector, using an iterator to denote the location. The member function that performs this task is erase(). There are two forms; the first takes a single iterator and removes an individual value, while the second takes a pair of iterators and removes all values in the given range. The size of the vector is reduced, but the capacity is unchanged. If the container type defines a destructor, the destructor will be invoked on the eliminated values.

```
vec_five.erase(where);
                                    // erase from the 12 to the end
where = find(vec_five.begin(), vec_five.end(), 12);
vec_five.erase(where, vec_five.end());
```

Iterator Invalidation

Once more, it is important to remember that should reallocation occur as a result of an insertion, all references, pointers, and iterators that denoted a location in the now-deleted memory block that held the values before reallocation become invalid.

5.2.6 Iteration

The member functions begin() and end() yield random access iterators for the container. Again, we note that the iterators yielded by these operations can become invalidated after insertions or removals of elements. The member functions rbegin() and rend() return similar iterators, however these access the underlying elements in reverse order. Constant iterators are returned if the original container is declared as constant, or if the target of the assignment or parameter is constant.

5.2.7 Test for Inclusion

A ***vector*** does not directly provide any method that can be used to determine if a specific value is contained in the collection. However, the generic algorithms find() or count() (Section 13.3.1 and 13.6.1) can be used for this purpose. The following statement, for example, tests to see whether an integer vector contains the element 17.

```
int num = 0;
count (vec_five.begin(), vec_five.end(), 17, num);

if (num)
   cout << "contains a 17" << endl;
else
   cout << "does not contain a 17" << endl;
```

Initializing Count

Note that count() returns its result through an argument that is passed by reference. It is important that this value be properly initialized before invoking this function.

5.2.8 Sorting and Sorted Vector Operations

A vector does not automatically maintain its values in sequence. However, a vector can be placed in order using the generic algorithm sort() (Section 14.2). The simplest form of sort uses for its comparisons the less-than operator for the element type. An alternative version of the generic algorithm permits the programmer to specify the comparison operator explicitly. This can be used, for example, to place the elements in descending rather than ascending order:

```
   // sort ascending
sort (aVec.begin(), aVec.end());

   // sort descending, specifying the ordering function explicitly
sort (aVec.begin(), aVec.end(), greater<int>() );

   // alternate way to sort descending
sort (aVec.rbegin(), aVec.rend());
```

A number of the operations described in Section 14 can be applied to a vector holding an ordered collection. For example, two vectors can be merged using the generic algorithm `merge()` (Section 14.6).

```
   // merge two vectors, printing output
merge (vecOne.begin(), vecOne.end(), vecTwo.begin(), vecTwo.end(),
   ostream_iterator<int,char> (cout, " "));
```

Sorting a vector also lets us use the more efficient binary search algorithms (Section 14.5), instead of a linear traversal algorithm such as `find()`.

5.2.9 Useful Generic Algorithms

Most of the algorithms described in Section 13 can be used with vectors. The following table summarizes a few of the more useful of these. For example, the maximum value in a vector can be determined as follows:

```
vector<int>::iterator where =
   max_element (vec_five.begin(), vec_five.end());
cout << "maximum is " << *where << endl;
```

Purpose	*Name*
Fill a vector with a given initial value	`fill`
Copy one sequence into another	`copy`
Copy values from a generator into a vector	`generate`
Find an element that matches a condition	`find`
Find consecutive duplicate elements	`adjacent_find`
Find a sub-sequence within a vector	`search`
Locate maximum or minimum element	`max_element, min_element`
Reverse order of elements	`reverse`
Replace elements with new values	`replace`
Rotate elements around a midpoint	`rotate`
Partition elements into two groups	`partition`
Generate permutations	`next_permutation`

Purpose	Name
Inplace merge within a vector	inplace_merge
Randomly shuffle elements in vector	random_shuffle
Count number of elements that satisfy condition	count
Reduce vector to a single value	accumulate
Inner product of two vectors	inner_product
Test two vectors for pair-wise equality	equal
Lexical comparison	lexicographical_compare
Apply transformation to a vector	transform
Partial sums of values	partial_sum
Adjacent differences of value	adjacent_difference
Execute function on each element	for_each

5.3 Boolean Vectors

Vectors of bit values (boolean 1/0 values) are handled as a special case by the standard library, so that the values can be efficiently packed (several elements to a word). The operations for a boolean vector , ***vector<bool>***, are a superset of those for an ordinary vector, only the implementation is more efficient.

One new member function added to the boolean vector data type is `flip()`. When invoked, this function inverts all the bits of the vector. Boolean vectors also return as reference an internal value that also supports the `flip()` member function.

```
vector<bool> bvec(27);
bvec.flip();                // flip all values
bvec[17].flip();            // flip bit 17
```

vector<bool> also supports an additional `swap()` member function that allows you to swap the values indicated by a pair of references.

```
bvec.swap(bvec [17], bvec [16]);
```

5.4 Example Program – Sieve of Eratosthenes

Obtaining the Source

Source for this program is found in the file `sieve.cpp`.

An example program that illustrates the use of vectors is the classic algorithm, called the *sieve of Eratosthenes*, used to discover prime numbers. A list of all the numbers up to some bound is represented by an integer vector. The basic idea is to strike out (set to zero) all those values that cannot be primes; thus all the remaining values will be the prime numbers. To do this, a loop examines each value in turn, and for those that are set to one (and thus have not yet been excluded from the set of candidate primes) strikes out all multiples of the number. When the outermost loop is finished, all remaining prime values have been discovered. The program is as follows:

```
void main() {
      // create a sieve of integers, initially set
   const int sievesize = 100;
   vector<int> sieve(sievesize, 1);

      // now search for 1 bit positions
   for (int i = 2; i * i < sievesize; i++)
      if (sieve[i])
         for (int j = i + i; j < sievesize; j += i)
            sieve[j] = 0;

      // finally, output the values that are set
   for (int j = 2; j < sievesize; j++)
      if (sieve[j])
         cout << j << " ";
   cout << endl;
}
```

Section 6. list

6.1 The list Data Abstraction

The ***vector*** data structure is a container of relatively fixed size. While the standard library provides facilities for dynamically changing the size of a vector, such operations are costly and should be used only rarely. Yet in many problems, the size of a collection may be difficult to predict in advance, or may vary widely during the course of execution. In such cases an alternative data structure should be employed. In this section we will examine an alternative data structure that can be used in these circumstances, the ***list*** data type.

A list corresponds to the intuitive idea of holding elements in a linear (although not necessarily ordered) sequence. New values can be added or removed either to or from the front of the list, or to or from the back. By using an iterator to denote a position, elements can also be added or removed to or from the middle of a list. In all cases the insertion or removal operations are efficient; they are performed in a constant amount of time that is independent of the number of elements being maintained in the collection. Finally, a list is a linear structure. The contents of the list cannot be accessed by subscript, and, in general, elements can only be accessed by a linear traversal of all values.

6.1.1 Include files

Whenever you use a ***list***, you must include the `list` header file.

```
# include <list>
```

6.2 List Operations

The member functions provided by the list data type are described in more detail below. Note that while member functions provide basic operations, the utility of the data structure is greatly extended through the use of the generic algorithms described in Sections 13 and 14.

6.2.1 Declaration and Initialization of Lists

Memory Management

Note that if you declare a container as holding pointers, you are responsible for managing the memory for the objects pointed to. The container classes will not, for example, automatically free memory for these objects when an item is erased from the container.

There are a variety of ways to declare a list. In the simplest form, a list is declared by simply stating the type of element the collection will maintain. This can be a primitive language type (such as `integer` or `double`), a pointer type, or a user-defined type. In the latter case, the user-defined type *must* implement a default constructor (a constructor with no arguments), as this constructor is in some cases used to initialize newly created elements. A collection declared in this fashion will initially not contain any elements.

```
list <int> list_one;
list <Widget *> list_two;
list <Widget> list_three;
```

An alternative form of declaration creates a collection that initially contains some number of equal elements. The constructor for this form is declared as `explicit`, meaning it cannot be used as a conversion operator. This prevents integers from inadvertently being converted into lists. The constructor for this form takes two arguments, a size and an initial value. The second argument is optional. If only the number of initial elements to be created is given, these values will be initialized with the default constructor; otherwise the elements will be initialized with the value of the second argument:

```
list <int> list_four (5);  // five elements, initialized to zero
list <double> list_five (4, 3.14);   // 4 values, initially 3.14
list <Widget> wlist_six (4);  // default constructor, 4 elements
list <Widget> list_six (3, Widget(7));  // 3 copies of Widget(7)
```

Lists can also be initialized using elements from another collection, using a beginning and ending iterator pair. The arguments can be any form of iterator, thus collections can be initialized with values drawn from any of the container classes in the standard library that support iterators. Because this requires the ability to specialize a member function using a template, some compilers may not yet support this feature. In these cases an alternative technique using the `copy()` generic algorithm can be employed. When a list is initialized using `copy()`, an *insert iterator* must be constructed to convert the output operations performed by the copy operation into list insertions. (See Section 2.4.) The inserter requires two arguments; the list into which the value is to be inserted, and an iterator indicating the location at which values will be placed. Insert iterators can also be used to copy elements into an arbitrary location in an existing list.

```
list <double> list_seven (aVector.begin(), aVector.end());

              // the following is equivalent to the above
list <double> list_eight;
copy (aVector.begin(), aVector.end(),
       inserter(list_eight, list_eight.begin()));
```

The `insert()` operation, to be described in Section 6.2.3, can also be used to place values denoted by an iterator into a list. Insert iterators can be used to initialize a list with a sequence of values produced by a *generator* (see Section 13.2.3). This is illustrated by the following:

```
list <int> list_nine;
                                // initialize list 1 2 3 ... 7
generate_n (inserter(list_nine, list_nine.begin()),
    7, iotaGen(1));
```

A *copy constructor* can be used to initialize a list with values drawn from another list. The assignment operator performs the same actions. In both cases the assignment operator for the element type is used to copy each new value.

```
list <int> list_ten (list_nine);          // copy constructor
list <Widget> list_eleven;
list_eleven = list_six;        // values copied by assignment
```

The `assign()` member function is similar to the assignment operator, but is more versatile and, in some cases, requires more arguments. Like an assignment, the existing values in the container are deleted, and replaced with the values specified by the arguments. If a destructor is provided for the container element type, it will be invoked for the elements being removed. There are two forms of `assign()`. The first takes two iterator arguments that specify a sub-sequence of an existing container. The values from this sub-sequence then become the new elements in the receiver. The second version of assign takes a count and an optional value of the container element type. After the call the container will hold the number of elements specified by the count, which will be equal to either the default value for the container type or the initial value specified.

```
list_six.assign(list_ten.begin(), list_ten.end());
list_four.assign(3, 7);          // three copies of value seven
list_five.assign(12);            // twelve copies of value zero
```

Finally, two lists can exchange their entire contents by means of the operation `swap()`. The argument container will take on the values of the receiver, while the receiver will assume those of the argument. A swap is very efficient, and should be used, where appropriate, in preference to an explicit element-by-element transfer.

```
list_ten.swap(list_nine);        // exchange lists nine and ten
```

6.2.2 Type Definitions

The class ***list*** includes a number of type definitions. The most common use for these is in declaration statements. For example, an iterator for a list of integers can be declared as follows:

```
list<int>::iterator location;
```

In addition to `iterator`, the following types are defined:

`value_type`	The type associated with the elements the list maintains.
`const_iterator`	An iterator that does not allow modification of

	the underlying sequence.
`reverse_iterator`	An iterator that moves in a backward direction.
`const_reverse_iterator`	A combination constant and reverse iterator.
`reference`	A reference to an underlying element.
`const_reference`	A reference to an underlying element that will not permit the element to be modified.
`size_type`	An unsigned integer type, used to refer to the size of containers.
`difference_type`	A signed integer type, used to describe distances between iterators.
`allocator_type`	The allocator type used for all storage management by the list.

6.2.3 Placing Elements into a List

Values can be inserted into a list in a variety of ways. Elements are most commonly added to the front or back of a list. These tasks are provided by the `push_front()` and `push_back()` operations, respectively. These operations are efficient (constant time) for both types of containers.

```
list_seven.push_front(1.2);
list_eleven.push_back (Widget(6));
```

In a previous discussion (Section 6.2.1) we noted how, with the aid of an insert iterator and the `copy()` or `generate()` generic algorithm, values can be placed into a list at a location denoted by an iterator. There is also a member function, named `insert(),` that avoids the need to construct the inserter. As we will describe shortly, the values returned by the iterator generating functions `begin()` and `end()` denote the beginning and end of a list, respectively. An insert using one of these is equivalent to `push_front()` or `push_back(),` respectively. If we specify only one iterator, the default element value is inserted.

```
// insert default type at beginning of list
list_eleven.insert(list_eleven.begin());
// insert widget 8 at end of list
list_eleven.insert(list_eleven.end(), Widget(8));
```

An iterator can denote a location in the middle of a list. There are several ways to produce this iterator. For example, we can use the result of any of the searching operations described in Section 13.3, such as an invocation of the `find()` generic algorithm. The new value is inserted immediately *prior* to the location denoted by the iterator. The `insert()` operation itself returns an iterator denoting the location of the inserted value. This result value was ignored in the invocations shown above.

```
//  find the location of the first occurrence of the
//     value 5 in list
list<int>::iterator location =
     find(list_nine.begin(), list_nine.end(), 5);
// and insert an 11 immediate before it
location = list_nine.insert(location, 11);
```

It is also possible to insert a fixed number of copies of an argument value. This form of `insert()` does not yield the location of the values.

```
line_nine.insert (location, 5, 12);       // insert five twelves
```

Finally, an entire sequence denoted by an iterator pair can be inserted into a list. Again, no useful value is returned as a result of the `insert()`.

```
// insert entire contents of list_ten into list_nine
list_nine.insert (location, list_ten.begin(), list_ten.end());
```

There are a variety of ways to *splice* one list into another. A splice differs from an insertion in that the item is simultaneously added to the receiver list and removed from the argument list. For this reason, a splice can be performed very efficiently, and should be used whenever appropriate. As with an insertion, the member function `splice()` uses an iterator to indicate the location in the receiver list where the splice should be made. The argument is either an entire list, a single element in a list (denoted by an iterator), or a sub-sequence of a list (denoted by a pair of iterators).

```
// splice the last element of list ten
list_nine.splice (location, list_ten, list_ten.end());
// splice all of list ten
list_nine.splice (location,  list_ten);
// splice list 9 back into list 10
list_ten.splice (list_ten.begin(), list_nine,
   list_nine.begin(), location);
```

Two ordered lists can be combined into one using the `merge()` operation. Values from the argument list are merged into the ordered list, leaving the argument list empty. The merge is stable; that is, elements retain their relative ordering from the original lists. As with the generic algorithm of the same name (Section 14.6), two forms are supported. The second form uses the binary function supplied as argument to order values. Not all compilers support the second form. If the second form is desired and not supported, the more general generic algorithm can be used, although this is slightly less efficient.

```
// merge with explicit compare function
list_eleven.merge(list_six, widgetCompare);
```

Iteration Invalidation

Unlike a **vector** *or* **deque**, *insertions or removals from the middle of a* **list** *will not invalidate references or pointers to other elements in the container. This property can be important if two or more iterators are being used to refer to the same container.*

```
   //the following is similar to the above
   list<Widget> list_twelve;
   merge (list_eleven.begin(), list_eleven.end(),
      list_six.begin(), list_six.end(),
      inserter(list_twelve, list_twelve.begin()), widgetCompare);
   list_eleven.swap(list_twelve);
```

6.2.4 Removing Elements

Just as there are a number of different ways to insert an element into a list, there are a variety of ways to remove values from a list. The most common operations used to remove a value are `pop_front()` or `pop_back()`, which delete the single element from the front or the back of the list, respectively. These member functions simply remove the given element, and do not themselves yield any useful result. If a destructor is defined for the element type it will be invoked as the element is removed. To look at the values before deletion, use the member functions `front()` or `back()`.

The `erase()` operation can be used to remove a value denoted by an iterator. For a list, the argument iterator, and any other iterators that denote the same location, become invalid after the removal, but iterators denoting other locations are unaffected. We can also use `erase()` to remove an entire subsequence, denoted by a pair of iterators. The values beginning at the initial iterator and up to, but not including, the final iterator are removed from the list. Erasing elements from the middle of a list is an efficient operation, unlike erasing elements from the middle of a vector or a deque.

```
   list_nine.erase (location);

   // erase values between the first occurrence of 5
   // and the following occurrence of 7
list<int>::iterator
   location = find(list_nine.begin(), list_nine.end(), 5);
   list<int>::iterator location2 =
       find(location, list_nine.end(), 7);
   list_nine.erase (location, location2);
```

The `remove()` member function removes all occurrences of a given value from a list. A variation, `remove_if()`, removes all values that satisfy a given predicate. An alternative to the use of either of these is to use the `remove()` or `remove_if()` generic algorithms (Section 13.5.1). The generic algorithms do not reduce the size of the list, instead they move the elements to be retained to the front of the list, leave the remainder of the list unchanged, and return an iterator denoting the location of the first unmodified element. This value can be used in conjunction with the `erase()` member function to remove the remaining values.

```
   list_nine.remove(4);                          // remove all fours
   list_nine.remove_if(divisibleByThree);     //remove any div by 3

   // the following is equivalent to the above
   list<int>::iterator location3 =
    remove_if(list_nine.begin(), list_nine.end(),
              divisibleByThree);
```

```
list_nine.erase(location3, list_nine.end());
```

The operation `unique()` will erase all but the first element from every consecutive group of equal elements in a list. The list need not be ordered. An alternative version takes a binary function, and compares adjacent elements using the function, removing the second value in those situations were the function yields a true value. As with `remove_if(),` not all compilers support the second form of `unique().` In this case the more general `unique()` generic algorithm can be used (see Section 13.5.2). In the following example the binary function is the greater-than operator, which will remove all elements smaller than a preceding element.

```
// remove first from consecutive equal elements
list_nine.unique();

// explicitly give comparison function
list_nine.unique(greater<int>());

// the following is equivalent to the above
location3 =
     unique(list_nine.begin(), list_nine.end(), greater<int>());
list_nine.erase(location3, list_nine.end());
```

6.2.5 Extent and Size-Changing Operations

The member function `size()` will return the number of elements being held by a container. The function `empty()` will return `true` if the container is empty, and is more efficient than comparing the size against the value `zero`.

```
cout << "Number of elements: " << list_nine.size () << endl;
if ( list_nine.empty () )
   cout << "list is empty " << endl;
else
   cout << "list is not empty " << endl;
```

The member function `resize()` changes the size of the list to the value specified by the argument. Values are either added or erased from the end of the collection as necessary. An optional second argument can be used to provide the initial value for any new elements added to the collection.

```
// become size 12, adding values of 17 if necessary
list_nine.resize (12, 17);
```

6.2.6 Access and Iteration

The member functions `front()` and `back()` return, but do not remove, the first and last items in the container, respectively. For a list, access to other elements is possible only by removing elements (until the desired element becomes the front or back) or through the use of iterators.

There are three types of iterators that can be constructed for lists. The functions `begin()` and `end()` construct iterators that traverse the list in forward order. For the ***list*** data type `begin()` and `end()` create bidirectional

iterators. The alternative functions `rbegin()` and `rend()` construct iterators that traverse in reverse order, moving from the end of the list to the front.

6.2.7 Test for Inclusion

The list data types do not directly provide any method that can be used to determine if a specific value is contained in the collection. However, either the generic algorithms `find()` or `count()` (Sections 13.3.1 and 13.6.1) can be used for this purpose. The following statements, for example, test to see whether an integer list contains the element 17.

```
int num = 0;
count(list_five.begin(), list_five.end(), 17, num);
if (num > 0)
   cout << "contains a 17" << endl;
else
   cout << "does not contain a 17" << endl;

if (find(list_five.begin(), list_five.end(), 17) != list_five.end())
   cout << "contains a 17" << endl;
else
   cout << "does not contain a 17" << endl;
```

6.2.8 Sorting and Sorted List Operations

The member function `sort()` places elements into ascending order. If a comparison operator other than `<` is desired, it can be supplied as an argument.

```
list_ten.sort ( );                  // place elements into sequence
list_twelve.sort (widgetCompare);       // sort with widget compare
                                        // function
```

Once a list has been sorted, a number of the generic algorithms for ordered collections can be used with lists. These are described in detail in Section 14.

6.2.9 Searching Operations

The various forms of searching functions described in Section 13.3, namely `find()`, `find_if()`, `adjacent find()`, `mismatch()`, `max_element()`, `min_element()` or `search()` can be applied to list. In all cases the result is an iterator, which can be dereferenced to discover the denoted element, or used as an argument in a subsequent operation.

6.2.10 In Place Transformations

A number of operations can be applied to lists in order to transform them in place. Some of these are provided as member functions. Others make use of some of the generic functions described in Section 13.

For a list, the member function `reverse()` reverses the order of elements in the list.

```
list_ten.reverse();                    // elements are now reversed
```

The generic algorithm `transform()` (Section 13.7.1) can be used to modify every value in a container, by simply using the same container as both input and as result for the operation. The following, for example, increments each element of a list by one. To construct the necessary unary function, the first argument of the binary integer addition function is bound to the value one. The version of `transform()` that manipulates two parallel sequences can be used in a similar fashion.

```
transform(list_ten.begin(), list_ten.end(),
   list_ten.begin(), bind1st(plus<int>(), 1));
```

Similarly, the functions `replace()` and `replace_if()` (Section 13.4.2) can be used to replace elements of a list with specific values. Rotations (Section 13.4.3) and partitions (Section 13.4.4), can also be performed with lists.

```
// find the location of the value 5, and rotate around it
location = find(list_ten.begin(), list_ten.end(), 5);
rotate(list_ten.begin(), location, list_ten.end());
// now partition using values greater than 7
partition(list_ten.begin(), list_ten.end(),
      bind2nd(greater<int>(), 7));
```

The functions `next_permutation()` and `prev_permutation()` (Section 13.4.5) can be used to generate the next permutation (or previous permutation) of a collection of values.

```
next_permutation (list_ten.begin(), list_ten.end());
```

6.2.11 Other Operations

The algorithm `for_each()` (Section 13.8.1) will apply a function to every element of a collection. An illustration of this use will be given in the radix sort example program in the section on the deque data structure.

The `accumulate()` generic algorithm reduces a collection to a scalar value (see Section 13.6.2). This can be used, for example, to compute the sum of a list of numbers. A more unusual use of `accumulate()` will be illustrated in the radix sort example.

```
cout << "Sum of list is: " <<
      accumulate(list_ten.begin(), list_ten.end(), 0) << endl;
```

Verify Search Results

The searching algorithms in the standard library will always return the end of range iterator if no element matching the search condition is found. Unless the result is guaranteed to be valid, it is a good idea to check for the end of range condition.

Two lists can be compared against each other. They are equal if they are the same size and all corresponding elements are equal. A list is less than another list if it is lexicographically smaller (see Section 13.6.5).

6.3 Example Program – An Inventory System

Obtaining the Sample Program

The executable version of the widget works program is contained in file `widwork.cpp` on the distribution disk.

We will use a simple inventory management system to illustrate the use of several list operations. Assume a business, named *WorldWideWidgetWorks*, requires a software system to manage their supply of widgets. Widgets are simple devices, distinguished by different identification numbers:

```
class  Widget {
public:
   Widget(int a = 0) : id(a) { }
   void operator = (const Widget& rhs) { id = rhs.id; }
   int id;
   friend ostream & operator << (ostream & out,const Widget & w)
      { return out << "Widget " << w.id; }
   friend bool operator == (const Widget& lhs, const Widget& rhs)
      { return lhs.id == rhs.id; }
   friend bool operator< (const Widget& lhs, const Widget& rhs)
      { return lhs.id < rhs.id; }
};
```

The state of the inventory is represented by two lists. One list represents the stock of widgets on hand, while the second represents the type of widgets that customers have backordered. The first is a list of widgets, while the second is a list of widget identification types. To handle our inventory we have two commands; the first, `order()`, processes orders, while the second, `receive()`, processes the shipment of a new widget.

```
class inventory {
public:
   void order (int wid);     // process order for widget type wid
   void receive (int wid);   // receive widget of type wid in
shipment
private:
   list<Widget> on_hand;
   list<int> on_order;
};
```

When a new widget arrives in shipment, we compare the widget identification number with the list of widget types on backorder. We use `find()` to search the backorder list, immediately shipping the widget if necessary. Otherwise it is added to the stock on hand.

```
void inventory::receive (int wid)
{
   cout << "Received shipment of widget type " << wid << endl;
   list<int>::iterator weneed =
      find (on_order.begin(), on_order.end(), wid);
   if (weneed != on_order.end())
   {
      cout << "Ship " << Widget(wid)
           << " to fill back order" << endl;
      on_order.erase(weneed);
   }
   else
      on_hand.push_front(Widget(wid));
}
```

When a customer orders a new widget, we scan the list of widgets in stock to determine if the order can be processed immediately. We can use the function `find_if()` to search the list. To do so we need a binary function that takes as its argument a widget and determines whether the widget matches the type requested. We can do this by taking a general binary widget-testing function, and binding the second argument to the specific widget type. To use the function `bind2nd(),` however, requires that the binary function be an instance of the class ***binary_function***. The general widget-testing function is written as follows:

```
class WidgetTester : public binary_function<Widget, int, bool> {
public:
   bool operator () (const Widget & wid, int testid) const
      { return wid.id == testid; }
};
```

The widget order function is then written as follows:

```
void inventory::order (int wid)
{
   cout << "Received order for widget type " << wid << endl;
   list<Widget>::iterator wehave =
         find_if (on_hand.begin(), on_hand.end(),
            bind2nd(WidgetTester(), wid));
   if (wehave != on_hand.end())
   {
      cout << "Ship " << *wehave << endl;
      on_hand.erase(wehave);
   }
   else
   {
      cout << "Back order widget of type "  << wid  << endl;
      on_order.push_front(wid);
   }
}
```

Section 7. deque

7.1 The deque Data Abstraction

The name "deque" is short for "double-ended queue," and is pronounced like "deck." Traditionally, the term is used to describe any data structure that permits both insertions and removals from either the front or the back of a collection. The ***deque*** container class permits this, as well as much more. In fact, the capabilities of the ***deque*** data structure are almost a union of those provided by the ***vector*** and ***list*** classes.

- Like a vector, the deque is an indexed collection. Values can be accessed by subscript, using the position within the collection as a key. (A capability not provided by the list class).
- Like a list, values can be efficiently added either to the front or to the back of a deque. (A capability provided only in part by the vector class).
- As with both the list and vector classes, insertions can be made into the middle of the sequence held by a deque. Such insertion operations are not as efficient as with a list, but slightly more efficient that they are in a vector.

In short, a deque can often be used both in situations that require a vector and in those that call for a list. Often, the use of a deque in place of either a vector or a list will result in faster programs. To determine which data structure should be used, you can refer to the set of questions described in Section 4.2

7.1.1 Include Files

The `deque` header file must appear in all programs that use the ***deque*** data type.

```
# include <deque>
```

7.2 Deque Operations

A ***deque*** is declared in the same fashion as a ***vector***, and includes within the class the same type definitions as vector.

The `begin()` and `end()` member functions return random access iterators, rather than bidirectional iterators, as they do for lists.

An insertion (either `insert()`, `push_front()`, or `push_back()`) can potentially invalidate all outstanding iterators and references to elements in the deque. As with the vector data type, this is a much more restrictive condition than insertions into a list.

If the underlying element type provides a destructor, then the destructor will be invoked when a value is erased from a deque.

Since the deque data type provides random access iterators, all the generic algorithms that operate with vectors can also be used with deques.

A vector holds elements in a single large block of memory. A deque, on the other hand, uses a number of smaller blocks. This may be important on systems that restrict the size of memory blocks, as it will permit a deque to hold many more elements than a vector.

As values are inserted, the index associated with any particular element in the collection will change. For example, if a value is inserted into position 3, then the value formerly indexed by 3 will now be found at index location 4, the value formerly at 4 will be found at index location 5, and so on.

7.3 Example Program – Radix Sort

The radix sort algorithm is a good illustration of how lists and deques can be combined with other containers. In the case of radix sort, a vector of deques is manipulated, much like a hash table.

Obtaining the Sample Program

The complete radix sort program is found in the file `radix.cpp` *in the tutorial distribution disk.*

Radix sorting is a technique for ordering a list of positive integer values. The values are successively ordered on digit positions, from right to left. This is accomplished by copying the values into "buckets," where the index for the bucket is given by the position of the digit being sorted. Once all digit positions have been examined, the list must be sorted.

The following table shows the sequences of values found in each bucket during the four steps involved in sorting the list 624 852 426 987 269 146 415 301 730 78 593. During pass 1 the ones place digits are ordered. During pass 2 the tens place digits are ordered, retaining the relative positions of values set by the earlier pass. On pass 3 the hundreds place digits are ordered, again retaining the previous relative ordering. After three passes the result is an ordered list.

bucket	pass 1	pass 2	pass 3
0	730	301	78
1	301	415	146
2	852	624, 426	269
3	593	730	301
4	624	146	415, 426
5	415	852	593
6	426, 146	269	624
7	987	78	730
8	78	987	852
9	269	593	987

The radix sorting algorithm is simple. A `while` loop is used to cycle through the various passes. The value of the variable `divisor` indicates which digit is currently being examined. A boolean flag is used to determine when execution should halt. Each time the while loop is executed a vector of deques is declared. By placing the declaration of this structure inside the while loop, it is reinitialized to empty each step. Each time the loop is executed, the values in the list are copied into the appropriate bucket by executing the function `copyIntoBuckets()` on each value. Once distributed into the buckets, the values are gathered back into the list by means of an accumulation.

```
void radixSort(list<unsigned int> & values)
{
   bool flag = true;
   int divisor = 1;

   while (flag) {
      vector< deque<unsigned int> > buckets(10);
      flag = false;
      for_each(values.begin(), values.end(),
            copyIntoBuckets(...));
      accumulate(buckets.begin(), buckets.end(),
            values.begin(), listCopy);
      divisor *= 10;
      }
}
```

The use of the function `accumulate()` here is slightly unusual. The “scalar” value being constructed is the list itself. The initial value for the accumulation is the iterator denoting the beginning of the list. Each bucket is processed by the following binary function:

```
list<unsigned int>::iterator
      listCopy(list<unsigned int>::iterator c,
         deque<unsigned int> & lst)
{
   // copy list back into original list, returning end
```

```
   return copy(lst.begin(), lst.end(), c);
}
```

The only difficulty remaining is defining the function `copyIntoBuckets()`. The problem here is that the function must take as its argument only the element being inserted, but it must also have access to the three values `buckets`, `divisor` and `flag`. In languages that permit functions to be defined within other functions the solution would be to define `copyIntoBuckets()` as a local function within the `while` loop. But C++ has no such facilities. Instead, we must create a class definition, which can be initialized with references to the appropriate values. The parenthesis operator for this class is then used as the function for the `for_each()` invocation in the radix sort program.

```
class copyIntoBuckets {
public:
   copyIntoBuckets
      (int d, vector< deque<unsigned int> > & b, bool & f)
         : divisor(d), buckets(b), flag(f) {}

   int divisor;
   vector<deque<unsigned int> > & buckets;
   bool & flag;

   void operator () (unsigned int v)
   {   int index = (v / divisor) % 10;
       // flag is set to true if any bucket
       // other than zeroth is used
       if (index) flag = true;
       buckets[index].push_back(v);
   }
};
```

Section 8.
set, multiset, and bitset

8.1 The set Data Abstraction

A *set* is a collection of values. Because the container used to implement the ***set*** data structure maintains values in an ordered representation, sets are optimized for insertion and removal of elements, and for testing to see whether a particular value is contained in the collection. Each of these operations can be performed in a logarithmic number of steps, whereas for a ***list***, ***vector***, or ***deque***, each operation requires in the worst case an examination of every element held by the container. For this reason, sets should be the data structure of choice in any problem that emphasizes insertion, removal, and test for inclusion of values. Like a ***list***, a ***set*** is not limited in size, but rather expands and contracts as elements are added to or removed from the collection.

There are two varieties of sets provided by the standard library. In the ***set*** container, every element is unique. Insertions of values that are already contained in the set are ignored. In the ***multiset*** container, on the other hand, multiple occurrences of the same value are permitted.

Sets, Ordered and Not

Although the abstract concept of a set does not necessarily imply an ordered collection, the **set** *data type is always ordered. If necessary, a collection of values that cannot be ordered can be maintained in, for example, a* **list**.

8.1.1 Include Files

Whenever you use a ***set*** or a ***multiset***, you must include the set header file.

```
# include <set>
```

8.2 set and multiset Operations

The member functions provided by the set and multiset data types will shortly be described in more detail. Note that while member functions provide basic operations, the utility of these data structures is greatly extended through the use of the generic algorithms described in Sections 13 and 14.

Sets and Bags

In other programming languages, a multiset is sometimes referred to as a bag.

8.2.1 Declaration and Initialization of Set

A ***set*** is a template data structure, specialized by the type of the elements it contains, and the operator used to compare keys. The latter argument is optional, and, if it is not provided, the less than operator for the key type will be assumed. The element type can be a primitive language type (such as integer or double), a pointer type, or a user-defined type. The element type must recognize both the equality testing operator (operator ==) and the less than comparison operator (operator <).

Initializing Sets with Iterators

As we noted in the earlier discussion on vectors and lists, the initialization of containers using a pair of iterators requires a mechanism that is still not widely supported by compilers. If not provided, the equivalent effect can be produced by declaring an empty set and then using the copy() generic algorithm to copy values into the set.

Sets can be declared with no initial elements, or they can be initialized from another container by providing a pair of iterators. An optional argument in both cases is an alternative comparison function; this value overrides the value provided by the template parameter. This mechanism is useful if a program contains two or more sets with the same values but different orderings, as it prevents more than one copy of the set member function from being instantiated. The copy constructor can be used to form a new set that is a clone, or copy, of an existing set.

```
set <int> set_one;

set <int, greater<int> > set_two;
set <int> set_three(greater<int>());

set <gadget, less<gadget> > gset;
set <gadget> gset(less<gadget>());

set <int> set_four (aList.begin(), aList.end());
set <int> set_five
   (aList.begin(), aList.end(), greater<int>());

set <int> set_six (set_four);                // copy constructor
```

A set can be assigned to another set, and two sets can exchange their values using the swap() operation (in a manner analogous to other standard library containers).

```
set_one = set_five;
set_six.swap(set_two);
```

8.2.2 Type Definitions

The classes ***set*** and ***multiset*** include a number of type definitions. The most common use for these is in a declaration statement. For example, an iterator for a set of integers can be declared in the following fashion:

```
set<int>::iterator location;
```

In addition to iterator, the following types are defined:

value_type	The type associated with the elements the set maintains.
const_iterator	An iterator that does not allow modification of the underlying sequence.
reverse_iterator	An iterator that moves in a backward direction.
const_reverse_iterator	A combination constant and reverse iterator.
reference	A reference to an underlying element.
const_reference	A reference to an underlying element that will not permit modification.
size_type	An unsigned integer type, used to refer to the

	size of containers.
`value_compare`	A function that can be used to compare two elements.
`difference_type`	A signed integer type, used to describe the distance between iterators.
`allocator_type`	An allocator used by the container or all storage management.

8.2.3 Insertion

Unlike a list or vector, there is only one way to add a new element to a ***set***. A value can be inserted into a set or a multiset using the `insert()` member function. With a multiset, the function returns an iterator that denotes the value just inserted. Insert operations into a ***set*** return a ***pair*** of values, in which the first field contains an iterator, and the second field contains a boolean value that is true if the element was inserted, and false otherwise. Recall that in a set, an element will not be inserted if it matches an element already contained in the collection.

The Pair Data Type

*If you want to use the **pair** data type without using maps, you should include the header file named `utility`.*

```
set_one.insert (18);

if (set_one.insert(18).second)
   cout << "element was inserted" << endl;
else
   cout << "element was not inserted " << endl;
```

Insertions of several elements from another container can also be performed using an iterator pair:

```
set_one.insert (set_three.begin(), set_three.end());
```

The ***pair*** data structure is a tuple of values. The first value is accessed through the field name `first`, while the second is, naturally, named `second`. A function named `make_pair()` simplifies the task of producing an instance of class ***pair***.

```
template <class T1, class T2>
struct pair {
   T1 first;
   T2 second;
   pair (const T1 & x, const T2 & y) : first(x), second(y) { }
};

template <class T1, class T2>
inline pair<T1, T2> make_pair(const T1& x, const T2& y)
   { return pair<T1, T2>(x, y); }
```

In determining the equivalence of keys, for example, to determine if the key portion of a new element matches any existing key, the comparison function for keys is used, and not the equivalence (`==`) operator. Two keys are deemed equivalent if the comparison function used to order key values

yields false in both directions. That is, if `Compare(key1, key2)` is false, and if `Compare(key2, key1)` is false, then `key1` and `key2` are considered equivalent.

8.2.4 Removal of Elements from a Set

Values are removed from a set using the member function `erase()`. The argument can be either a specific value, an iterator that denotes a single value, or a pair of iterators that denote a range of values. When the first form is used on a multiset, all arguments matching the argument value are removed, and the return value indicates the number of elements that have been erased.

```
// erase element equal to 4
set_three.erase(4);

// erase element five
set<int>::iterator five = set_three.find(5);
set_three.erase(five);

// erase all values between seven and eleven
set<int>::iterator seven = set_three.find(7);
set<int>::iterator eleven = set_three.find(11);
set_three.erase (seven, eleven);
```

If the underlying element type provides a destructor, then the destructor will be invoked prior to removing the element from the collection.

8.2.5 Searching and Counting

The member function `size()` will yield the number of elements held by a container. The member function `empty()` will return a boolean true value if the container is empty, and is generally faster than testing the size against zero.

The member function `find()` takes an element value, and returns an iterator denoting the location of the value in the set if it is present, or a value matching the end-of-set (the value yielded by the function `end()`) if it is not. If a multiset contains more than one matching element, the value returned can be any appropriate value.

```
set<int>::iterator five = set_three.find(5);
if (five != set_three.end())
    cout << "set contains a five" << endl;
```

The member functions `lower_bound()` and `upper_bound()` are most useful with multisets, as with sets they simply mimic the function `find()`. The member function `lower_bound()` yields the first entry that matches the argument key, while the member function `upper_bound()` returns the first value past the last entry matching the argument. Finally, the member function `equal_range()` returns a ***pair*** of iterators, holding the lower and upper bounds.

The member function count() returns the number of elements that match the argument. For a set this value is either zero or one, whereas for a multiset it can be any nonnegative value. Since a non-zero integer value is treated as true, the count() function can be used to test for inclusion of an element, if all that is desired is to determine whether or not the element is present in the set. The alternative, using find(), requires testing the result returned by find() against the end-of-collection iterator.

```
if (set_three.count(5))
   cout << "set contains a five" << endl;
```

8.2.6 Iterators

The member functions begin() and end() produce iterators for both sets and multisets. The iterators produced by these functions are constant to ensure that the ordering relation for the set is not inadvertently or intentionally destroyed by assigning a new value to a set element. Elements are generated by the iterators in sequence, ordered by the comparison operator provided when the set was declared. The member functions rbegin() and rend() produce iterators that yield the elements in reverse order.

No Iterator Invalidation

Unlike a vector or deque, the insertion or removal of values from a set does not invalidate iterators or references to other elements in the collection.

8.2.7 Set Operations

The traditional set operations of subset test, set union, set intersection, and set difference are not provided as member functions, but are instead implemented as generic algorithms that will work with any ordered structure. These functions are described in more detail in Section 14.7. The following summary describes how these functions can be used with the set and multiset container classes.

8.2.7.1 Subset test

The function includes() can be used to determine if one set is a subset of another; that is, if all elements from the first are contained in the second. In the case of multisets the number of matching elements in the second set must exceed the number of elements in the first. The four arguments are a pair of iterators representing the (presumably) smaller set, and a pair of iterators representing the (potentially) larger set.

```
if (includes(set_one.begin(), set_one.end(),
   set_two.begin(), set_two.end()))
      cout << "set_one is a subset of set_two" << endl;
```

The less than operator (operator <) will be used for the comparison of elements, regardless of the operator used in the declaration of the set. Where this is inappropriate, an alternative version of the includes() function is provided. This form takes a fifth argument, which is the comparison function used to order the elements in the two sets.

8.2.7.2 Set Union or Intersection

The function `set_union()` can be used to construct a union of two sets. The two sets are specified by iterator pairs, and the union is copied into an output iterator that is supplied as a fifth argument. To form the result as a set, an *insert iterator* must be used to form the output iterator. (See Section 2.4 for a discussion of insert iterators.) If the desired outcome is a union of one set with another, then a temporary set can be constructed, and the results swapped with the argument set prior to deletion of the temporary set.

```
// union two sets, copying result into a vector
vector<int> v_one (set_one.size() + set_two.size());

set_union(set_one.begin(), set_one.end(),
   set_two.begin(), set_two.end(), v_one.begin());

// form union in place
set<int> temp_set;
set_union(set_one.begin(), set_one.end(),
   set_two.begin(), set_two.end(),
   inserter(temp_set, temp_set.begin()));
set_one.swap(temp_set);  // temp_set will be deleted
```

The function `set_intersection()` is similar, and forms the intersection of the two sets.

As with the `includes()` function, the less than operator (operator `<`) is used to compare elements in the two argument sets, regardless of the operator provided in the declaration of the sets. Should this be inappropriate, alternative versions of both the `set_union()` or `set_intersection()` functions permit the comparison operator used to form the set to be given as a sixth argument.

The operation of taking the union of two multisets should be distinguished from the operation of merging two sets. Imagine that one argument set contains three instances of the element 7, and the second set contains two instances of the same value. The union will contain only three such values, while the merge will contain five. To form the merge, the function `merge()` can be used (see Section 14.6). The arguments to this function exactly match those of the `set_union()` function.

8.2.7.3 Set Difference

There are two forms of set difference. A simple set difference represents the elements in the first set that are not contained in the second. A symmetric set difference is the union of the elements in the first set that are not contained in the second, with the elements in the second that are not contained in the first. These two values are constructed by the functions `set_difference()` and `set_symmetric_difference()`, respectively. The use of these functions is similar to the use of the `set_union()` function described earlier.

8.2.8 Other Generic Algorithms

Because sets are ordered and have constant iterators, a number of the generic functions described in Sections 13 and 14 either are not applicable to sets or are not particularly useful. However, the following table gives a few of the functions that can be used in conjunction with the ***set*** data type.

Purpose	*Name*	*Section*
Copy one sequence into another	`copy`	13.2.2
Find an element that matches a condition	`find_if`	13.3.1
Find a sub-sequence within a set	`search`	13.3.3
Count number of elements that satisfy condition	`count_if`	13.6.1
Reduce set to a single value	`accumulate`	13.6.2
Execute function on each element	`for_each`	13.8.1

8.3 Example Program: – A Spelling Checker

A simple example program that uses a set is a spelling checker. The checker takes as arguments two input streams; the first representing a stream of correctly spelled words (that is, a dictionary), and the second a text file. First, the dictionary is read into a set. This is performed using a `copy()` and an input stream iterator, copying the values into an inserter for the dictionary. Next, words from the text are examined one by one, to see if they are in the dictionary. If they are not, then they are added to a set of misspelled words. After the entire text has been examined, the program outputs the list of misspelled words.

Obtaining the Sample Program

This program can be found in the file `spell.cpp` in the tutorial distribution.

```
void spellCheck (istream & dictionary, istream & text)
{
   typedef set <string, less<string> > stringset;
   stringset words, misspellings;
   string word;
   istream_iterator<string, ptrdiff_t> dstream(dictionary), eof;

   // first read the dictionary
   copy (dstream, eof, inserter(words, words.begin()));

   // next read the text
   while (text >> word)
      if (! words.count(word))
         misspellings.insert(word);

   // finally, output all misspellings
   cout << "Misspelled words:" << endl;
   copy (misspellings.begin(), misspellings.end(),
      ostream_iterator<string>(cout, "\n"));
}
```

An improvement would be to suggest alternative words for each misspelling. There are various heuristics that can be used to discover alternatives. The technique we will use here is to simply exchange adjacent letters. To find these, a call on the following function is inserted into the loop that displays the misspellings.

```
void findMisspell(stringset & words, string & word)
{
   for (int I = 1; I < word.length(); I++) {
      swap(word[I-1], word[I]);
      if (words.count(word))
         cout << "Suggestion: " << word << endl;
      // put word back as before
      swap(word[I-1], word[I]);
      }
}
```

8.4 The bitset Abstraction

A ***bitset*** is really a cross between a ***set*** and a ***vector***. Like the vector abstraction ***vector<bool>***, the abstraction represents a set of binary (0/1 bit) values. However, set operations can be performed on bitsets using the logical bit-wise operators. The class ***bitset*** does not provide any iterators for accessing elements.

8.4.1 Include Files

```
#include <bitset>
```

8.4.2 Declaration and Initialization of bitset

A ***bitset*** is a template class abstraction. The template argument is not, however, a type, but an integer value. The value represents the number of bits the set will contains.

```
bitset<126> bset_one;        // create a set of 126 bits
```

An alternative technique permits the size of the set to be specified as an argument to the constructor. The actual size will be the smaller of the value used as the template argument and the constructor argument. This technique is useful when a program contains two or more bit vectors of differing sizes. Consistently using the larger size for the template argument means that only one set of methods for the class will be generated. The actual size, however, will be determined by the constructor.

```
bitset<126> bset_two(100);   // this set has only 100 elements
```

A third form of constructor takes as argument a string of 0 and 1 characters. A bitset is created that has as many elements as are characters in the string, and is initialized with the values from the string.

```
bitset<126> small_set("10101010");   // this set has 8 elements
```

8.4.3 Accessing and Testing Elements

An individual bit in the bitset can be accessed using the subscript operation. Whether the bit is one or not can be determined using the member function `test()`. Whether any bit in the bitset is "on" is tested using the member function `any()`, which yields a boolean value. The inverse of `any()` is returned by the member function `none()`.

```
bset_one[3] = 1;
if (bset_one.test(4))
   cout << "bit position 4 is set" << endl;
if (bset_one.any())
   cout << "some bit position is set" << endl;
if (bset_one.none()) cout << "no bit position is set" << endl;
```

The function `set()` can be used to set a specific bit. `bset_one.set(I)` is equivalent to `bset_one[I] = true`. Invoking the function without any arguments sets all bit positions to true. The function `reset()` is similar, and sets the indicated positions to false (sets all positions to false if invoked with no argument). The function `flip()` flips either the indicated position, or all positions if no argument is provided. The function `flip()` is also provided as a member function for the individual bit references.

```
bset_one.flip();   // flip the entire set
bset_one.flip(12);   // flip only bit 12
bset_one[12].flip();   // reflip bit 12
```

The member function `size()` returns the size of the bitset, while the member function `count()` yields the number of bits that are set.

8.4.4 Set operations

Set operations on bitsets are implemented using the bit-wise operators, in a manner analogous to the way in which the same operators act on integer arguments.

The negation operator (operator `~`) applied to a bitset returns a new bitset containing the inverse of elements in the argument set.

The intersection of two bitsets is formed using the `and` operator (operator `&`). The assignment form of the operator can be used. In the assignment form, the target becomes the disjunction of the two sets.

```
bset_three = bset_two & bset_four;
bset_five &= bset_three;
```

The union of two sets is formed in a similar manner using the *or* operator (operator `|`). The exclusive-or is formed using the bit-wise exclusive or operator (operator `^`).

The left and right shift operators (operator `<<` and `>>`) can be used to shift a bitset left or right, in a manner analogous to the use of these operators on integer arguments. If a bit is shifted left by an integer value *n*, then the new bit position *I* is the value of the former *I-n*. Zeros are shifted into the new positions.

8.4.5 Conversions

The member function `to_ulong()` converts a ***bitset*** into an `unsigned long`. It is an error to perform this operation on a bitset containing more elements than will fit into this representation.

The member function `to_string()` converts a ***bitset*** into an object of type ***string.*** The string will have as many characters as the bitset. Each zero bit will correspond to the character `0`, while each one bit will be represented by the character `1`.

Section 9.

map and multimap

9.1 The map Data Abstraction

A ***map*** is an indexed data structure, similar to a ***vector*** or a ***deque***. However, maps differ from vectors or deques in two important respects. First, in a map, unlike a vector or deque, the index values (called the *key values*) need not be integer, but can be any ordered data type. For example, maps can be indexed by real numbers, or by strings. Any data type for which a comparison operator can be defined can be used as a key. As with a ***vector*** or ***deque***, elements can be accessed through the use of the subscript operator (although there are other techniques). The second important difference is that a map is an ordered data structure. This means that elements are maintained in sequence, the ordering being determined by key values. Because they maintain values in order, maps can very rapidly find the element specified by any given key (searching is performed in logarithmic time). Like a ***list***, maps are not limited in size, but expand or contract as necessary as new elements are added or removed. In large part, a ***map*** can simply be considered to be a ***set*** that maintains a collection of pairs.

Other Names for Maps

In other programming languages, a map-like data structure is sometimes referred to as a dictionary, *a* table, *or an* associative array.

There are two varieties of maps provided by the standard library. The ***map*** data structure demands unique keys. That is, there is a one-to-one association between key elements and their corresponding value. In a map, the insertion of a new value that uses an existing key is ignored. A ***multimap***, on the other hand, permits multiple different entries to be indexed by the same key. Both data structures provide relatively fast (logarithmic time) insertion, deletion, and access operations.

9.1.1 Include files

Whenever you use a ***map*** or a ***multimap***, you must include the `map` header file.

```
# include <map>
```

Pairs

See the discussion of insertion in Section 8 for a description of the pair data type.

9.2 Map and Multimap Operations

The member functions provided by the map and multimap data types will shortly be described in more detail. Note that while member functions provide basic operations, the utility of the data structure is greatly extended through the use of the generic algorithms described in Sections 13 and 14.

9.2.1 Declaration and Initialization of map

The declaration of a map follows the pattern we have seen repeatedly in the standard library. A map is a template data structure, specialized by the type of the key elements, the type of the associated values, and the operator to be used in comparing keys. If your compiler supports default template types (a

relatively new feature in C++ not yet supported by all vendors), then the last of these is optional, and if not provided, the less than operator for the key type will be assumed. Maps can be declared with no initial elements, or initialized from another container by providing a pair of iterators. In the latter case the iterators must denote values of type ***pair***; the first field in each pair is taken to be a key, while the second field is a value. A copy constructor also permits maps to be created as copies of other maps.

```
// map indexed by doubles containing strings
map<double, string, less<double> > map_one;
// map indexed by integers, containing integers
map<int, int> map_two(aContainer.begin(), aContainer.end());
// create a new map, initializing it from map two
map<int, int> map_three (map_two);   // copy constructor
```

A map can be assigned to another map, and two maps can exchange their values using the `swap()` operation (in a manner analogous to other standard library containers).

9.2.2 Type Definitions

The classes ***map*** and ***multimap*** include a number of type definitions. These are most commonly used in declaration statements. For example, an iterator for a map of strings to integers can be declared in the following fashion:

```
map<string, int>::iterator location;
```

In addition to `iterator`, the following types are defined:

`key_type`	The type associated with the keys used to index the map.
`value_type`	The type held by the container, a key/value pair.
`const_iterator`	An iterator that does not allow modification of the underlying sequence.
`reverse_iterator`	An iterator that moves in a backward direction.
`const_reverse_iterator`	A combination constant and reverse iterator.
`reference`	A reference to an underlying value.
`const_reference`	A reference to an underlying value that will not permit the element to be modified.
`size_type`	An unsigned integer type, used to refer to the size of containers.
`key_compare`	A function object that can be used to compare two keys.
`value_compare`	A function object that can be used to compare two elements.

`difference_type`	A signed integer type, used to describe the distances between iterators.
`allocator_type`	An allocator used by the container for all storage management.

9.2.3 Insertion and Access

Values can be inserted into a map or a multimap using the `insert()` operation. Note that the argument must be a key-value pair. This pair is often constructed using the data type `value_type` associated with the map.

```
map_three.insert (map<int>::value_type(5, 7));
```

Insertions can also be performed using an iterator pair, for example as generated by another map.

```
map_two.insert (map_three.begin(), map_three.end());
```

With a map (but not a multimap), values can be accessed and inserted using the subscript operator. Simply using a key as a subscript creates an entry – the default element is used as the associated value. Assigning to the result of the subscript changes the associated binding.

```
cout << "Index value 7 is " << map_three[7] << endl;
   // now change the associated value
map_three[7] = 5;
cout << "Index value 7 is " << map_three[7] << endl;
```

9.2.4 Removal of Values

Values can be removed from a map or a multimap by naming the key value. In a multimap the erasure removes all elements with the associated key. An element to be removed can also be denoted by an iterator; as, for example, the iterator yielded by a `find()` operation. A pair of iterators can be used to erase an entire range of elements.

```
// erase the 4th element 4
map_three.erase(4);
// erase the 5th element
mtesttype::iterator five = map_three.find(5);
map_three.erase(five);

// erase all values between the 7th and 11th elements
mtesttype::iterator seven = map_three.find(7);
mtesttype::iterator eleven = map_three.find(11);
map_three.erase (seven, eleven);
```

If the underlying element type provides a destructor, then the destructor will be invoked prior to removing the key and value pair from the collection.

9.2.5 Iterators

No Iterator Invalidation

Unlike a vector or deque, the insertion or removal of elements from a map does not invalidate iterators which may be referencing other portions of the container.

The member functions `begin()` and `end()` produce bidirectional iterators for both maps and multimaps. Dereferencing an iterator for either a map or a multimap will yield a ***pair*** of key / value elements. The field names `first` and `second` can be applied to these values to access the individual fields. The first field is constant, and cannot be modified. The second field, however, can be used to change the value being held in association with a given key. Elements will be generated in sequence, based on the ordering of the key fields.

The member functions `rbegin()` and `rend()` produce iterators that yield the elements in reverse order.

9.2.6 Searching and Counting

The member function `size()` will yield the number of elements held by a container. The member function `empty()` will return a boolean true value if the container is empty, and is generally faster than testing the size against zero.

The member function `find()` takes a key argument, and returns an iterator denoting the associated key / value pair. In the case of multimaps, the first such value is returned. In both cases the past-the-end iterator is returned if no such value is found.

```
if (map_one.find(4) != map_one.end())
   cout << "contains a 4th element" << endl;
```

The member function `lower_bound()` yields the first entry that matches the argument key, while the member function `upper_bound()` returns the first value past the last entry matching the argument. Finally, the member function `equal_range()` returns a ***pair*** of iterators, holding the lower and upper bounds. An example showing the use of these procedures will be presented later in this section.

The member function `count()` returns the number of elements that match the key value supplied as the argument. For a map, this value is always either zero or one, whereas for a multimap it can be any nonnegative value. If you simply want to determine whether or not a collection contains an element indexed by a given key, using `count()` is often easier than using the `find()` function and testing the result against the end-of-sequence iterator.

```
if (map_one.count(4))
   cout << "contains a 4th element" << endl;
```

9.2.7 Element Comparisons

The member functions `key_comp()` and `value_comp()`, which take no arguments, return function objects that can be used to compare elements of the key or value types. Values used in these comparisons need not be

contained in the collection, and neither function will have any effect on the container.

```
if (map_two.key_comp (i, j))
   cout << "element i is less than j" << endl;
```

9.2.8 Other Map Operations

Because maps and multimaps are ordered collections, and because the iterators for maps return pairs, many of the functions described in Sections 13 and 14 are meaningless or difficult to use. However, there are a few notable exceptions. The functions `for_each()`, `adjacent_find()`, and `accumulate()` each have their own uses. In all cases it is important to remember that the functions supplied as arguments should take a key/value pair as arguments.

9.3 Example Programs

We present three example programs that illustrate the use of maps and multimaps. These are a telephone database, graphs, and a concordance.

9.3.1 A Telephone Database

A maintenance program for a simple telephone database is a good application for a map. The database is simply an indexed structure, where the name of the person or business (a string) is the key value, and the telephone number (a long) is the associated entry. We might write such a class as follows:

Obtaining the Sample Program

The complete example program is included in the file tutorial `tele.cpp` in the distribution.

```
typedef map<string, long, less<string> > friendMap;
typedef friendMap::value_type entry_type;

class telephoneDirectory {
public:
   void addEntry (string name, long number)   // add new entry to
                                              // database
      { database[name] = number; }

   void remove (string name)   // remove entry from database
      { database.erase(name); }

   void update (string name, long number)   // update entry
      { remove(name); addEntry(name, number); }

   void displayDatabase()   // display entire database
      { for_each(database.begin(), database.end(), printEntry); }

   void displayPrefix(int);   // display entries that match prefix

   void displayByPrefix();   // display database sorted by prefix

private:
   friendMap database;
};
```

Simple operations on our database are directly implemented by map commands. Adding an element to the database is simply an `insert`, removing an element is an `erase`, and updating is a combination of the two. To print all the entries in the database we can use the `for_each()` algorithm, and apply the following simple utility routine to each entry:

```
void printEntry(const entry_type & entry)
   { cout << entry.first << ":" << entry.second << endl; }
```

We will use a pair of slightly more complex operations to illustrate how a few of the algorithms described in Section 13 can be used with maps. Suppose we wanted to display all the phone numbers with a certain three digit initial prefix[1] . We will use the `find_if()` function (which is different from the `find()` member function in class map) to locate the first entry. Starting from this location, subsequent calls on `find_if()` will uncover each successive entry.

```
void telephoneDirectory::displayPrefix(int prefix)
{
   cout << "Listing for prefix " << prefix << endl;
   friendMap::iterator where;
   where =
      find_if (database.begin(), database.end(),
             checkPrefix(prefix));
   while (where != database.end()) {
      printEntry(*where);
      where = find_if (++where, database.end(),
             checkPrefix(prefix));
      }
   cout << "end of prefix listing" << endl;
}
```

For the predicate to this operation, we require a boolean function that takes only a single argument (the pair representing a database entry), and tells us whether or not it is in the given prefix. There is no obvious candidate function, and in any case the test prefix is not being passed as an argument to the comparison function. The solution to this problem is to employ a technique that is commonly used with the standard library, defining the predicate function as an instance of a class, and storing the test predicate as an instance variable in the class, initialized when the class is constructed. The desired function is then defined as the function call operator for the class:

```
int prefix(const entry_type & entry)
   { return entry.second / 10000; }

class checkPrefix {
public:
   checkPrefix (int p) : testPrefix(p) { }
   int testPrefix;
```

---

[1] We apologize to international readers for this obviously North-American-centric example.

```
   bool operator () (const entry_type & entry)
      { return prefix(entry) == testPrefix; }
};
```

Our final example will be to display the directory sorted by prefix. It is not possible to alter the order of the maps themselves. So instead, we create a new map with the element types reversed, then copy the values into the new map, which will order the values by prefix. Once the new map is created, it is then printed.

```
typedef map<long, string, less<long> > sortedMap;
typedef sortedMap::value_type sorted_entry_type;

void telephoneDirectory::displayByPrefix()
{
   cout << "Display by prefix" << endl;
   sortedMap sortedData;
   friendMap::iterator itr;
   for (itr = database.begin(); itr != database.end(); itr++)
      sortedData.insert(sortedMap::value_type((*itr).second,
            (*itr).first));
   for_each(sortedData.begin(), sortedData.end(),
            printSortedEntry);
}
```

The function used to print the sorted entries is the following:

```
void printSortedEntry (const sorted_entry_type & entry)
      { cout << entry.first << ":" << entry.second << endl; }
```

9.3.2 Graphs

A ***map*** whose elements are themselves ***map***s are a natural representation for a directed graph. For example, suppose we use strings to encode the names of cities, and we wish to construct a map where the value associated with an edge is the distance between two connected cities. We could create such a graph as follows:

Obtaining the Sample Program

The executable version of this program is found in the file `graph.cpp` *on the tutorial distribution disk.*

```
typedef map<string, int> stringVector;
typedef map<string, stringVector> graph;

const string pendleton("Pendleton");   // define strings for
                                       // city names
const string pensacola("Pensacola");
const string peoria("Peoria");
const string phoenix("Phoenix");
const string pierre("Pierre");
const string pittsburgh("Pittsburgh");
const string princeton("Princeton");
const string pueblo("Pueblo");

graph cityMap;      // declare the graph that holds the map

cityMap[pendleton][phoenix] = 4;   // add edges to the graph
cityMap[pendleton][pueblo] = 8;
cityMap[pensacola][phoenix] = 5;
cityMap[peoria][pittsburgh] = 5;
cityMap[peoria][pueblo] = 3;
cityMap[phoenix][peoria] = 4;
```

```
cityMap[phoenix][pittsburgh] = 10;
cityMap[phoenix][pueblo] = 3;
cityMap[pierre][pendleton] = 2;
cityMap[pittsburgh][pensacola] = 4;
cityMap[princeton][pittsburgh] = 2;
cityMap[pueblo][pierre] = 3;
```

The type ***stringVector*** is a map of integers indexed by strings. The type ***graph*** is, in effect, a two-dimensional sparse array, indexed by strings and holding integer values. A sequence of assignment statements initializes the graph.

A number of classic algorithms can be used to manipulate graphs represented in this form. One example is Dijkstra's shortest-path algorithm. Dijkstra's algorithm begins from a specific city given as an initial location. A ***priority_queue*** of distance/city pairs is then constructed, and initialized with the distance from the starting city to itself (namely, zero). The definition for the distance pair data type is as follows:

```
struct DistancePair {
   unsigned int first;
   string second;
   DistancePair() : first(0) { }
   DistancePair(unsigned int f, const string & s)
      : first(f), second(s) { }
};

bool operator < (const DistancePair & lhs, const DistancePair & rhs)
   { return lhs.first < rhs.first; }
```

In the algorithm that follows, note how the conditional test is reversed on the priority queue, because at each step we wish to pull the smallest, and not the largest, value from the collection. On each iteration around the loop we pull a city from the queue. If we have not yet found a shorter path to the city, the current distance is recorded, and by examining the graph we can compute the distance from this city to each of its adjacent cities. This process continues until the priority queue becomes exhausted.

```
void shortestDistance(graph & cityMap,
      const string & start, stringVector & distances)
{
   // process a priority queue of distances to cities
   priority_queue<DistancePair, vector<DistancePair>,
                  greater<DistancePair> > que;
   que.push(DistancePair(0, start));

   while (! que.empty()) {
      // pull nearest city from queue
      int distance = que.top().first;
      string city = que.top().second;
      que.pop();
            // if we haven't seen it already, process it
      if (0 == distances.count(city)) {
               // then add it to shortest distance map
         distances[city] = distance;
               // and put values into queue
         const stringVector & cities = cityMap[city];
         stringVector::const_iterator start = cities.begin();
```

```
         stringVector::const_iterator stop = cities.end();
         for (; start != stop; ++start)
            que.push(DistancePair(distance + (*start).second,
                                  (*start).first));
         }
      }
}
```

Notice that this relatively simple algorithm makes use of ***vector***s, ***map***s, ***string***s and ***priority_queue***s. ***priority_queue***s are described in greater detail in Section 11.

9.3.3 A Concordance

A concordance is an alphabetical listing of words in a text, that shows the line numbers on which each word occurs. We develop a concordance to illustrate the use of the ***map*** and ***multimap*** container classes. The data values will be maintained in the concordance by a multimap, indexed by strings (the words) and will hold integers (the line numbers). A multimap is employed because the same word will often appear on multiple different lines; indeed, discovering such connections is one of the primary purposes of a concordance. Another possibility would have been to use a ***map*** and use a ***set*** of integer elements as the associated values.

Obtaining the Sample Program

An executable version of the concordance program is found on the tutorial distribution file under the name `concord.cpp`.

```
class concordance {
   typedef multimap<string, int less <string> > wordDictType;
public:
   void addWord (string, int);
   void readText (istream &);
   void printConcordance (ostream &);

private:
   wordDictType wordMap;
};
```

The creation of the concordance is divided into two steps: first the program generates the concordance (by reading lines from an input stream), and then the program prints the result on the output stream. This is reflected in the two member functions `readText()` and `printConcordance()`. The first of these, `readText()`, is written as follows:

```
void concordance::readText (istream & in)
{
   string line;
   for (int i = 1; getline(in, line, '\n'); i++) {
      allLower(line);
      list<string> words;
      split (line, " ,.;:", words);
      list<string>::iterator wptr;
      for (wptr = words.begin(); wptr != words.end(); ++wptr)
         addWord(*wptr, i);
      }
}
```

Lines are read from the input stream one by one. The text of the line is first converted into lower case, then the line is split into words, using the function

split() described in Section 12.3. Each word is then entered into the concordance. The method used to enter a value into the concordance is as follows:

```
void concordance::addWord (string word, int line)
{
   // see if word occurs in list
   // first get range of entries with same key
   wordDictType::iterator low = wordMap.lower_bound(word);
   wordDictType::iterator high = wordMap.upper_bound(word);
   // loop over entries, see if any match current line
   for ( ; low != high; ++low)
      if ((*low).second == line)
         return;
   // didn't occur, add now
   wordMap.insert(wordDictType::value_type(word, line));
}
```

The major portion of addWord() is concerned with ensuring values are not duplicated in the word map if the same word occurs twice on the same line. To assure this, the range of values matching the key is examined, each value is tested, and if any match the line number then no insertion is performed. It is only if the loop terminates without discovering the line number that the new word/line number pair is inserted.

The final step is to print the concordance. This is performed in the following fashion:

```
void concordance::printConcordance (ostream & out)
{
   string lastword("");
   wordDictType::iterator pairPtr;
   wordDictType::iterator stop = wordMap.end();
   for (pairPtr = wordMap.begin(); pairPtr != stop; ++pairPtr)
         // if word is same as previous, just print line number
      if (lastword == (*pairPtr).first)
         out << " " << (*pairPtr).second;
      else {   // first entry of word
         lastword = (*pairPtr).first;
         cout << endl << lastword << ": " << (*pairPtr).second;
         }
   cout << endl; // terminate last line
}
```

An iterator loop is used to cycle over the elements being maintained by the word list. Each new word generates a new line of output – thereafter line numbers appear separated by spaces. If, for example, the input was the text:

```
It was the best of times,
it was the worst of times.
```

The output, from best to worst, would be:

```
best: 1
it: 1 2
of: 1 2
the: 1 2
```

```
times: 1 2
was: 1 2
worst: 1
```

Section 10.
stack and queue

10.1 Overview

Most people have a good intuitive understanding of the ***stack*** and ***queue*** data abstractions, based on experience with everyday objects. An excellent example of a stack is a pile of papers on a desk, or a stack of dishes in a cupboard. In both cases the important characteristic is that it is the item on the top that is most easily accessed. The easiest way to add a new item to the collection is to place it above all the current items in the stack. In this manner, an item removed from a stack is the item that has been most recently inserted into the stack; for example, the top piece of paper in the pile, or the top dish in the stack.

An everyday example of a queue, on the other hand, is a bank teller line, or a line of people waiting to enter a theater. Here new additions are made to the back of the queue, as new people enter the line, while items are removed from the front of the structure, as patrons enter the theater. The removal order for a queue is the opposite of that for a stack. In a queue, the item that is removed is the element that has been present in the queue for the longest period of time.

LIFO and FIFO

A stack is sometimes referred to as a LIFO structure, and a queue is called a FIFO structure. The abbreviation LIFO stands for Last In, First Out. This means the first entry removed from a stack is the last entry that was inserted. The term FIFO, on the other hand, is short for First In, First Out. This means the first element removed from a queue is the first element that was inserted into the queue.

In the standard library, both stacks and queues are *adaptors*, built on top of other containers which are used to actually hold the values. A ***stack*** can be built out of a ***vector***, a ***list***, or a ***deque***, while a ***queue*** can be built on top of either a ***list*** or a ***deque***. Elements held by either a stack or queue must recognize both the operators `<` and `==`.

Because neither ***stacks*** nor ***queues*** define iterators, it is not possible to examine the elements of the collection except by removing the values one by one. The fact that these structures do not implement iterators also implies that most of the generic algorithms described in Sections 12 and 13 cannot be used with either data structure.

10.2 The stack Data Abstraction

As a data abstraction, a stack is traditionally defined as any object that implements the following operations:

`empty()`	return true if the collection is empty
`size()`	return number of elements in collection
`top()`	return (but do not remove) the topmost element in the stack
`push(newElement)`	push a new element onto the stack
`pop()`	remove (but do not return) the topmost element from the stack

10.2.1 Include Files

Note that accessing the front element and removing the front element are separate operations. Programs that use the stack data abstraction should include the file `stack`, as well as the include file for the container type (e.g., `vector`).

```
# include <stack>
# include <vector>
```

Right Angle Brackets

Note that on most compilers it is important to leave a space between the two right angle brackets in the declaration of a stack; otherwise they are interpreted by the compiler as a right shift operator.

10.2.2 Declaration and Initialization of stack

A declaration for a stack must specify two arguments; the underlying element type, and the container that will hold the elements. For a stack, the most common container is a ***vector*** or a ***deque***, however a ***list*** can also be used. The vector version is generally smaller, while the deque version may be slightly faster. The following are sample declarations for a stack.

```
stack< int, vector<int> > stackOne;
stack< double, deque<double> > stackTwo;
stack< Part *, list<Part * > > stackThree;
stack< Customer, list<Customer> > stackFour;
```

The last example creates a stack of a programmer-defined type named Customer.

10.2.3 Example Program – A RPN Calculator

A classic application of a stack is in the implementation of calculator. Input to the calculator consists of a text string that represents an expression written in reverse polish notation (RPN). Operands, that is, integer constants, are pushed on a stack of values. As operators are encountered, the appropriate number of operands are popped off the stack, the operation is performed, and the result is pushed back on the stack.

We can divide the development of our stack simulation into two parts, a calculator engine and a calculator program. A calculator engine is concerned with the actual work involved in the simulation, but does not perform any input or output operations. The name is intended to suggest an analogy to a car engine, or a computer processor – the mechanism performs the actual work, but the user of the mechanism does not normally directly interact with it. Wrapped around this is the calculator program, which interacts with the user, and passes appropriate instructions to the calculator engine.

Obtaining the Sample Program

This program is found in the file `calc.cpp` in the distribution package.

We can use the following class definition for our calculator engine. Inside the class declaration we define an enumerated list of values to represent each of the possible operators that the calculator is prepared to accept. We have made two simplifying assumptions: all operands will be integer values, and we will handle only binary operators.

```
class calculatorEngine {
public:
   enum binaryOperator {plus, minus, times, divide};
```

```
   int currentMemory ()           // return current top of stack
      { return data.top(); }

   void pushOperand (int value)   // push operand value on to stack
      { data.push (value); }

   void doOperator (binaryOperator);   // pop stack and perform
                                       // operator

protected:
   stack< int, vector<int> > data;
};
```

The member function `doOperator()` performs the actual work. It pops values from the stack, performs the operation, then pushes the result back onto the stack.

Defensive Programming

A more robust program would check to see if the stack was empty before attempting to perform the `pop()` operation.

```
void calculatorEngine::doOperator (binaryOperator theOp)
{
   int right = data.top();   // read top element
   data.pop();   // pop it from stack
   int left = data.top();   // read next top element
   data.pop();   // pop it from stack
   switch (theOp) {
      case plus: data.push(left + right); break;
      case minus: data.push(left - right); break;
      case times: data.push(left * right); break;
      case divide: data.push(left / right); break;
      }
}
```

The main program reads values in reverse polish notation, invoking the calculator engine to do the actual work:

```
void main() {
   int intval;
   calculatorEngine calc;
   char c;

   while (cin >> c)
      switch (c) {
         case '0': case '1': case '2': case '3': case '4':
         case '5': case '6': case '7': case '8': case '9':
            cin.putback(c);
            cin >> intval;
            calc.pushOperand(intval);
            break;

         case '+':  calc.doOperator(calculatorEngine::plus);
            break;

         case '-':  calc.doOperator(calculatorEngine::minus);
            break;

         case '*':  calc.doOperator(calculatorEngine::times);
            break;

         case '/':  calc.doOperator(calculatorEngine::divide);
            break;

         case 'p':  cout << calc.currentMemory() << endl;
```

```
            break;
         case 'q':  return; // quit program
      }
}
```

10.3 The queue Data Abstraction

As a data abstraction, a ***queue*** is traditionally defined as any object that implements the following operations:

`empty()`	return true if the collection is empty
`size()`	return number of elements in collection
`front()`	return (but do not remove) the element at the front of the queue
`back()`	return the element at the end of the queue
`push(newElement)`	push a new element on to the end of the queue
`pop()`	remove (but do not return) the element at the front of the queue

10.3.1 Include Files

Note that the operations of accessing and of removing the front elements are performed separately. Programs that use the queue data abstraction should include the file `queue`, as well as the include file for the container type (e.g., `list`).

```
# include <queue>
# include <list>
```

10.3.2 Declaration and Initialization of queue

A declaration for a queue must specify both the element type as well as the container that will hold the values. For a queue the most common containers are a ***list*** or a ***deque***. The list version is generally smaller, while the deque version may be slightly faster. The following are sample declarations for a queue.

```
queue< int, list<int> > queueOne;
queue< double, deque<double> > queueTwo;
queue< Part *, list<Part * > > queueThree;
queue< Customer, list<Customer> > queueFour;
```

The last example creates a queue of a programmer-defined type named Customer. As with the stack container, all objects stored in a queue must understand the operators `<` and `==`.

Because the queue does not implement an iterator, none of the generic algorithms described in Sections 12 or 13 apply to queues.

10.3.3 Example Program – Bank Teller Simulation

Queues are often found in businesses, such as supermarkets or banks. Suppose you are the manager of a bank, and you need to determine how many tellers to have working during certain hours. You decide to create a computer simulation, basing your simulation on certain observed behavior. For example, you note that during peak hours there is a ninety percent chance that a customer will arrive every minute.

Obtaining the Sample Program

The complete version of the bank teller simulation program is found in file `teller.cpp` *on the distribution disk.*

We create a simulation by first defining objects to represent both customers and tellers. For customers, the information we wish to know is the average amount of time they spend waiting in line. Thus, customer objects simply maintain two integer data fields: the time they arrive in line, and the time they will spend at the counter. The latter is a value randomly selected between 2 and 8. (See Section 2.2.5 for a discussion of the `randomInteger()` function.)

```
class Customer {
public:
   Customer (int at = 0) : arrival_Time(at),
         processTime(2 + randomInteger(6)) {}
   int arrival_Time;
   int processTime;

   bool done()      // are we done with our transaction?
      { return --processTime < 0; }

   operator < (const Customer & c)   // order by arrival time
      { return arrival_Time < c.arrival_Time; }

   operator == (const Customer & c)   // no two customers are alike
      { return false; }
};
```

Because objects can only be stored in standard library containers if they can be compared for equality and ordering, it is necessary to define the `<` and `==` operators for customers. Customers can also tell us when they are done with their transactions.

Tellers are either busy servicing customers, or they are free. Thus, each teller value holds two data fields; a customer, and a boolean flag. Tellers define a member function to answer whether they are free or not, as well as a member function that is invoked when they start servicing a customer.

```
class Teller {
public:
   Teller() { free = true; }

   bool isFree()   // are we free to service new customer?
      { if (free) return true;
        if (customer.done())
           free = true;
        return free;
```

```
      }

   void addCustomer(Customer c)   // start serving new customer
      {   customer = c;
         free = false;
      }

private:
   bool free;
   Customer customer;
};
```

The main program is then a large loop, cycling once each simulated minute. Each minute a new customer is, with probability 0.9, entered into the queue of waiting customers. Each teller is polled, and if any are free they take the next customer from the queue. Counts are maintained of the number of customers serviced and the total time they spent in queue. From these two values we can determine, following the simulation, the average time a customer spent waiting in the line.

```
void main() {
   int numberOfTellers = 5;
   int numberOfMinutes = 60;
   double totalWait = 0;
   int numberOfCustomers = 0;
   vector<Teller> teller(numberOfTellers);
   queue< Customer, deque<Customer> > line;

   for (int time = 0; time < numberOfMinutes; time++) {
      if (randomInteger(10) < 9)
         line.push(Customer(time));
      for (int i = 0; i < numberOfTellers; i++) {
         if (teller[i].isFree() & ! line.empty()) {
            Customer & frontCustomer = line.front();
            numberOfCustomers++;
            totalWait += (time - frontCustomer.arrival_Time);
            teller[i].addCustomer(frontCustomer);
            line.pop();
            }
         }
      }
   cout << "average wait:" <<
          (totalWait / numberOfCustomers) << endl;
}
```

By executing the program several times, using various values for the number of tellers, the manager can determine the smallest number of tellers that can service the customers while maintaining the average waiting time at an acceptable amount.

Section 11. priority_queue

11.1 The priority queue Data Abstraction

A ***priority queue*** is a data structure useful in problems where you need to rapidly and repeatedly find and remove the largest element from a collection of values. An everyday example of a priority queue is the "to do" list of tasks waiting to be performed that most of us maintain to keep ourselves organized. Some jobs, such as "clean desktop," are not imperative and can be postponed arbitrarily. Other tasks, such as "finish report by Monday" or "buy flowers for anniversary," are time-crucial and must be addressed more rapidly. Thus, we sort the tasks waiting to be accomplished in order of their importance (or perhaps based on a combination of their critical importance, their long term benefit, and the fun we will have doing them) and choose the most pressing.

A more computer-related example of a priority queue is that used by an operating system to maintain a list of pending processes, where the value associated with each element is the priority of the job. For example, it may be necessary to respond rapidly to a key pressed at a terminal, before the data is lost when the next key is pressed. On the other hand, the process of copying a listing to a queue of output waiting to be handled by a printer is something that can be postponed for a short period, as long as it is handled eventually. By maintaining processes in a priority queue, those jobs with urgent priority will be executed prior to any jobs with less urgent requirements.

Simulation programs use a priority queue of "future events." The simulation maintains a virtual "clock," and each event has an associated time when the event will take place. In such a collection, the element with the smallest time value is the next event that should be simulated. These are only a few instances of the types of problems for which a priority queue is a useful tool. You probably have, or will, encounter others.

A Queue That is Not a Queue

The term priority queue *is a misnomer, in that the data structure is not a queue, in the sense that we used the term in Section 10, since it does not return elements in a strict first-in, first-out sequence. Nevertheless, the name is now firmly associated with this particular data type.*

11.1.1 Include Files

Programs that use the priority queue data abstraction should include the file queue, as well as the include file for the container type (e.g., vector).

```
# include <queue>
# include <vector>
```

11.2 The Priority Queue Operations

A priority queue is a data structure that can hold elements of type `T` and that implements the following five operations:

`push(T)`	add a new value to the collection being maintained
`top()`	return a reference to the smallest element in collection
`pop()`	delete the smallest element from the collection
`size()`	return the number of elements in the collection
`empty()`	return true if the collection is empty

Elements of type `T` must be comparable to each other, either through the use of the default less than operator (the `<` operator), or through a comparison function passed either as a template argument or as an optional argument on the constructor. The latter form will be illustrated in the example program provided later in this section. As with all the containers in the Standard Library, there are two constructors. The default constructor requires either no arguments or the optional comparison function. An alternative constructor takes an iterator pair, and initializes the values in the container from the argument sequence. Once more, an optional third argument can be used to define the comparison function.

The priority queue data type is built on top of a container class, which is the structure actually used to maintain the values in the collection. There are two containers in the standard library that can be used to construct priority queues: ***vectors*** or ***deques***.

Initializing Queues from other containers

As we noted in earlier sections, support for initializing containers using a pair of iterators requires a feature that is not yet widely supported by compilers. While we document this form of constructor, it may not yet be available on your system.

11.2.1 Declaration and Initialization of priority queue

The following illustrates the declaration of several priority queues:

```
priority_queue< int, vector<int> > queue_one;
priority_queue< int, vector<int>, greater<int> > queue_two;
priority_queue< double, deque<double> >
      queue_three(aList.begin(), aList.end());
priority_queue< eventStruct, vector<eventStruct> >
      queue_four(eventComparison);
priority_queue< eventStruct, deque<eventStruct> >
      queue_five(aVector.begin(), aVector.end(), eventComparison);
```

Queues constructed out of vectors tend to be somewhat smaller, while queues constructed out of deques can be somewhat faster, particularly if the number of elements in the queue varies widely over the course of execution. However, these differences are slight, and either form will generally work in most circumstances.

Because the priority queue data structure does not itself know how to construct iterators, very few of the algorithms noted in Section 13 can be

used with priority queues. Instead of iterating over values, a typical algorithm that uses a priority queue constructs a loop, which repeatedly pulls values from the structure (using the `top()` and `pop()` operations) until the collection becomes empty (tested using the `empty()` operation). The example program described in the next section will illustrate this use.

Priority queues are implemented by internally building a data structure called a *heap*. Abstractly, a heap is a binary tree in which every node possesses the property that the value associated with the node is smaller than or equal to the value associated with either child node.

Information on Heaps. *Details of the algorithms used in manipulating heaps will not be discussed here, however such information is readily available in almost any textbook on data structures.*

11.3 Application – Event-Driven Simulation

An extended example will illustrate the use of priority queues. The example illustrates one of the more common uses for priority queues, which is to support the construction of a simulation model.

A *discrete event-driven simulation* is a popular simulation technique. Objects in the simulation model objects in the real world, and are programmed to react as much as possible as the real objects would react. A priority queue is used to store a representation of "events" that are waiting to happen. This queue is stored in order, based on the time the event should occur, so the smallest element will always be the next event to be modeled. As an event occurs, it can spawn other events. These subsequent events are placed into the queue as well. Execution continues until all events have been processed.

Finding Smallest Elements

We describe the priority queue as a structure for quickly discovering the largest *element in a sequence. If, instead, your problem requires the discovery of the* smallest *element, there are various possibilities. One is to supply the inverse operator as either a template argument or the optional comparison function argument to the constructor. If you are defining the comparison argument as a function, as in the example problem, another solution is to simply invert the comparison test.*

Events can be represented as subclasses of a base class, which we will call ***event***. The base class simply records the time at which the event will take place. A pure virtual function named `processEvent` will be invoked to execute the event.

```
class event {
public:
   event (unsigned int t) : time(t) { }
   const unsigned int time;
   virtual void processEvent() = 0;
};
```

The simulation queue will need to maintain a collection of different types of events. Each different form of event will be represented by a different subclass of class ***event***. Not all events will have the same exact type, although they will all be subclasses of class ***event***. (This is sometimes called a *heterogeneous* collection.) For this reason the collection must store *pointers* to events, instead of the events themselves. (In theory one could store references, instead of pointers, however the standard library containers cannot hold references).

Since comparison of pointers cannot be specialized on the basis of the pointer types, we must instead define a new comparison function for pointers to events. In the standard library this is accomplished by defining a new structure, the sole purpose of which is to define the function invocation operator (the `()` operator) in the appropriate fashion. Since in this particular example we wish to use the priority queue to return the *smallest* element each time, rather than the largest, the order of the comparison is reversed, as follows:

```
struct eventComparison {
   bool operator () (event * left, event * right) const
      { return left->time > right->time; }
};
```

We are now ready to define the class ***simulation***, which provides the structure for the simulation activities. The class ***simulation*** provides two functions. The first is used to insert a new event into the queue, while the second runs the simulation. A data field is also provided to hold the current simulation "time."

```
class simulation {
public:
   simulation () : eventQueue(), time(0) { }

   void scheduleEvent (event * newEvent)
      { eventQueue.push (newEvent); }

   void run();

   unsigned int time;

protected:
   priority_queue<event *, vector<event *>, eventComparison>
eventQueue;
};
```

Notice the declaration of the priority queue used to hold the pending events. In this case we are using a ***vector*** as the underlying container. We could just as easily have used a ***deque***.

The heart of the simulation is the member function `run()`, which defines the event loop. This procedure makes use of three of the five priority queue operations, namely `top()`, `pop()`, and `empty()`. It is implemented as follows:

```
void simulation::run()
{
   while (! eventQueue.empty()) {
      event * nextEvent = eventQueue.top();
      eventQueue.pop();
      time = nextEvent->time;
      nextEvent->processEvent();
      delete nextEvent;   // free memory used by event
      }
}
```

Storing Pointers versus Storing Values

Other example programs in this tutorial have all used containers to store values. In this example the container will maintain pointers to values, not the values themselves. Note that a consequence of this is that the programmer is then responsible for managing the memory for the objects being manipulated.

11.3.1 An Ice Cream Store Simulation

To illustrate the use of our simulation framework, this example program gives a simple simulation of an ice cream store. Such a simulation might be used, for example, to determine the optimal number of chairs that should be provided, based on assumptions such as the frequency that customers will arrive, the length of time they will stay, and so on.

Our store simulation will be based around a subclass of class ***simulation***, defined as follows:

```
class storeSimulation : public simulation {
public:
   storeSimulation()
      : freeChairs(35), profit(0.0), simulation() { }

   bool canSeat (unsigned int numberOfPeople);
   void order(unsigned int numberOfScoops);
   void leave(unsigned int numberOfPeople);

private:
   unsigned int freeChairs;
```

Obtaining the sample program

The complete event simulation is found in the file `icecream.cpp` on the distribution disk.

```
   double profit;
} theSimulation;
```

There are three basic activities associated with the store. These are arrival, ordering and eating, and leaving. This is reflected not only in the three member functions defined in the simulation class, but in three separate subclasses of ***event***.

The member functions associated with the store simply record the activities taking place, producing a log that can later be studied to evaluate the simulation.

```
bool storeSimulation::canSeat (unsigned int numberOfPeople)
   // if sufficient room, then seat customers
{
   cout << "Time: " << time;
   cout << " group of " << numberOfPeople << " customers arrives";
   if (numberOfPeople < freeChairs) {
      cout << " is seated" << endl;
      freeChairs -= numberOfPeople;
      return true;
      }
   else {
      cout << " no room, they leave" << endl;
      return false;
      }
}

void storeSimulation::order (unsigned int numberOfScoops)
   // serve icecream, compute profits
{
   cout << "Time: " << time;
   cout << " serviced order for " << numberOfScoops << endl;
   profit += 0.35 * numberOfScoops;
}

void storeSimulation::leave (unsigned int numberOfPeople)
   // people leave, free up chairs
{
   cout << "Time: " << time;
   cout << " group of size " << numberOfPeople <<
         " leaves" << endl;
   freeChairs += numberOfPeople;
}
```

As we noted already, each activity is matched by a subclass of event. Each subclass of ***event*** includes an integer data field, which represents the size of a group of customers. The arrival event occurs when a group enters. When executed, the arrival event creates and installs a new instance of order event. The function `randomInteger()` (see Section 2.2.5) is used to compute a random integer between 1 and the argument value.

```
class arriveEvent : public event {
public:
   arriveEvent (unsigned int time, unsigned int groupSize)
      : event(time), size(groupSize) { }
   virtual void processEvent ();
private:
   unsigned int size;
};
```

```
void arriveEvent::processEvent()
{               // see if everybody can be seated
   if (theSimulation.canSeat(size))
      theSimulation.scheduleEvent
         (new orderEvent(time + 1 + randomInteger(4), size));
}
```

An order event similarly spawns a leave event.

```
class orderEvent : public event {
public:
   orderEvent (unsigned int time, unsigned int groupSize)
      : event(time), size(groupSize) { }
   virtual void processEvent ();
private:
   unsigned int size;
};

void orderEvent::processEvent()
{               // each person orders some number of scoops
   for (int i = 0; i < size; i++)
      theSimulation.order(1 + rand(3));
   theSimulation.scheduleEvent
      (new leaveEvent(time + 1 + randomInteger(10), size));
};
```

Finally, leave events free up chairs, but do not spawn any new events.

```
class leaveEvent : public event {
public:
   leaveEvent (unsigned int time, unsigned int groupSize)
      : event(time), size(groupSize) { }
   virtual void processEvent ();
private:
   unsigned int size;
};

void leaveEvent::processEvent ()
{               // leave and free up chairs
   theSimulation.leave(size);
}
```

To run the simulation we simply create some number of initial events (say, 30 minutes worth), then invoke the `run()` member function.

```
void main() {
   // load queue with some number of initial events
   unsigned int t = 0;
   while (t < 30) {
      t += rand(6);
      theSimulation.scheduleEvent(
         new arriveEvent(t, 1 + randomInteger(4)));
      }

   // then run simulation and print profits
   theSimulation.run();
   cout << "Total profits " << theSimulation.profit << endl;
}
```

Section 12. String

12.1 The string Abstraction

A *string* is basically an indexable sequence of characters. In fact, although a string is not declared as a subclass of ***vector***, almost all of the vector operators discussed in Section 5 can be applied to string values. However, a string is also a much more abstract quantity, and, in addition to simple vector operators, the ***string*** data type provides a number of useful and powerful high level operations.

In the standard library, a string is actually a template class, named ***basic_string***. The template argument represents the type of character that will be held by the string container. By defining strings in this fashion, the standard library not only provides facilities for manipulating sequences of normal 8-bit ASCII characters, but also for manipulating other types of character-like sequences, such as 16-bit wide characters. The data types ***string*** and ***wstring*** (for wide string) are simply typedefs of ***basic_string***, defined as follows:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

As we have already noted, a string is similar in many ways to a vector of characters. Like the ***vector*** data type, there are two sizes associated with a string. The first represents the number of characters currently being stored in the string. The second is the *capacity*, the maximum number of characters that can potentially be stored into a string without reallocation of a new internal buffer. As it is in the vector data type, the capacity of a string is a dynamic quantity. When string operations cause the number of characters being stored in a string value to exceed the capacity of the string, a new internal buffer is allocated and initialized with the string values, and the capacity of the string is increased. All this occurs behind the scenes, requiring no interaction with the programmer.

Strings and Wide Strings

In the remainder of this section we will refer to the string data type, however all the operations we will introduce are equally applicable to wide strings.

12.1.1 Include Files

Programs that use strings must include the `string` header file:

```
# include <string>
```

12.2 String Operations

In the following sections, we'll examine the standard library operations used to create and manipulate strings.

12.2.1 Declaration and Initialization of string

The simplest form of declaration for a string simply names a new variable, or names a variable along with the initial value for the string. This form was used extensively in the example graph program given in Section 9.3.2. A copy constructor also permits a string to be declared that takes its value from a previously defined string.

```
string s1;
string s2 ("a string");
string s3 = "initial value";
string s4 (s3);
```

In these simple cases the capacity is initially exactly the same as the number of characters being stored. Alternative constructors let you explicitly set the initial capacity. Yet another form allows you to set the capacity and initialize the string with repeated copies of a single character value.

```
string s6 ("small value", 100);// holds 11 values, can hold 100
string s7 (10, '\n');          // holds ten newline characters
```

Initializing from Iterators

Remember, the ability to initialize a container using a pair of iterators requires the ability to declare a template member function using template arguments independent of those used to declare the container. At present not all compilers support this feature.

Finally, like all the container classes in the standard library, a string can be initialized using a pair of iterators. The sequence being denoted by the iterators must have the appropriate type of elements.

```
string s8 (aList.begin(), aList.end());
```

12.2.2 Resetting Size and Capacity

As with the ***vector*** data type, the current size of a string is yielded by the `size()` member function, while the current capacity is returned by `capacity()`. The latter can be changed by a call on the `reserve()` member function, which (if necessary) adjusts the capacity so that the string can hold at least as many elements as specified by the argument. The member function `max_size()` returns the maximum string size that can be allocated. Usually this value is limited only by the amount of available memory.

```
cout << s6.size() << endl;
cout << s6.capacity() << endl;
s6.reserve(200);                    // change capacity to 200
cout << s6.capacity() << endl;
cout << s6.max_size() << endl;
```

The member function `length()` is simply a synonym for `size()`. The member function `resize()` changes the size of a string, either truncating characters from the end or inserting new characters. The optional second argument for `resize()` can be used to specify the character inserted into the newly created character positions.

```
s7.resize(15, '\t');                // add tab characters at end
cout << s7.length() << endl;        // size should now be 15
```

The member function empty() returns true if the string contains no characters, and is generally faster than testing the length against a zero constant.

```
if (s7.empty())
   cout << "string is empty" << endl;
```

12.2.3 Assignment, Append and Swap

A string variable can be assigned the value of either another string, a literal C-style character array, or an individual character.

```
s1 = s2;
s2 = "a new value";
s3 = 'x';
```

The operator += can also be used with any of these three forms of argument, and specifies that the value on the right hand side should be *appended* to the end of the current string value.

```
s3 += "yz";                    // s3 is now xyz
```

The more general assign() and append() member functions let you specify a subset of the right hand side to be assigned to or appended to the receiver two arguments, pos and n, indicate that the n values following position pos should be assigned / appended.

```
s4.assign (s2, 0, 3);         // assign first three characters
s4.append (s5, 2, 3);         // append characters 2, 3 and 4
```

The addition operator + is used to form the catenation of two strings. The + operator creates a copy of the left argument, then appends the right argument to this value.

```
cout << (s2 + s3) << endl;    // output catenation of s2 and s3
```

As with all the containers in the standard library, the contents of two strings can be exchanged using the swap() member function.

```
s5.swap (s4);                 // exchange s4 and s5
```

12.2.4 Character Access

An individual character from a string can be accessed or assigned using the subscript operator. The member function at() is almost a synonym for this operation except an out_of_range exception will be thrown if the requested location is greater than or equal to size().

```
cout << s4[2] << endl;        // output position 2 of s4
s4[2] = 'x';                  // change position 2
cout << s4.at(2) << endl;     // output updated value
```

The member function c_str() returns a pointer to a null terminated character array, whose elements are the same as those contained in the string.

This lets you use strings with functions that require a pointer to a conventional C-style character array. The resulting pointer is declared as constant, which means that you cannot use `c_str()` to modify the string. In addition, the value returned by `c_str()` might not be valid after any operation that may cause reallocation (such as `append()` or `insert()`). The member function `data()` returns a pointer to the underlying character buffer.

```
char d[256];
strcpy(d, s4.c_str());                // copy s4 into array d
```

12.2.5 Iterators

The member functions `begin()` and `end()` return beginning and ending random-access iterators for the string. The values denoted by the iterators will be individual string elements. The functions `rbegin()` and `rend()` return backwards iterators.

Invalidating Iterators

Note that the contents of an iterator are not guaranteed to be valid after any operation that might force a reallocation of the internal string buffer, such as an append or an insertion.

12.2.6 Insertion, Removal and Replacement

The ***string*** member functions `insert()` and `erase()` are similar to the ***vector*** functions `insert()` and `erase()`. Like the vector versions, they can take iterators as arguments, and specify the insertion or removal of the ranges specified by the arguments. The function `replace()` is a combination of erase and insert, in effect replacing the specified range with new values.

```
s2.insert(s2.begin()+2, aList.begin(), aList.end());
s2.erase(s2.begin()+3, s2.begin()+5);
s2.replace(s2.begin()+3, s2.begin()+6, s3.begin(), s3.end());
```

In addition, the functions also have non-iterator implementations. The `insert()` member function takes as argument a position and a string, and inserts the string into the given position. The erase function takes two integer arguments, a position and a length, and removes the characters specified. And the replace function takes two similar integer arguments as well as a string and an optional length, and replaces the indicated range with the string (or an initial portion of a string, if the length has been explicitly specified).

```
s3.insert (3, "abc");      // insert abc after position 3
s3.erase (4, 2);           // remove positions 4 and 5
s3.replace (4, 2, "pqr");  // replace positions 4 and 5 with pqr
```

12.2.7 Copy and Substring

The member function `copy()` generates a substring then assigns this substring to the `char*` target given as the first argument. The range of values for the substring is specified either by an initial position, or a position and a length.

```
s3.copy (s4, 2);       // assign to s4 positions 2 to end of s3
s5.copy (s4, 2, 3);    // assign to s4 positions 2 to 4 of s5
```

The member function `substr()` returns a string that represents a portion of the current string. The range is specified by either an initial position, or a position and a length.

```
cout << s4.substr(3) << endl;       // output 3 to end
cout << s4.substr(3, 2) << endl;    // output positions 3 and 4
```

12.2.8 String Comparisons

The member function `compare()` is used to perform a lexical comparison between the receiver and an argument string. Optional arguments permit the specification of a different starting position or a starting position and length of the argument string. See Section 13.6.5 for a description of lexical ordering. The function returns a negative value if the receiver is lexicographically smaller than the argument, a zero value if they are equal and a positive value if the receiver is larger than the argument.

The relational and equality operators (`<`, `<=`, `==`, `!=`, `>=` and `>`) are all defined using the comparison member function. Comparisons can be made either between two strings, or between strings and ordinary C-style character literals.

Comparing Strings

Although the function is accessible, users will seldom invoke the member function `compare()` directly. Instead, comparisons of strings are usually performed using the conventional comparison operators, which in turn make use of the function `compare()`.

12.2.9 Searching Operations

The member function `find()` determines the first occurrence of the argument string in the current string. An optional integer argument lets you specify the starting position for the search. (Remember that string index positions begin at zero.) If the function can locate such a match, it returns the starting index of the match in the current string. Otherwise, it returns a value out of the range of the set of legal subscripts for the string. The function `rfind()` is similar, but scans the string from the end, moving backwards.

```
s1 = "mississippi";
cout << s1.find("ss") << endl;              // returns 2
cout << s1.find("ss", 3) << endl;           // returns 5
cout << s1.rfind("ss") << endl;             // returns 5
cout << s1.rfind("ss", 4) << endl;          // returns 2
```

The functions `find_first_of()`, `find_last_of()`, `find_first_not_of()`, and `find_last_not_of()` treat the argument string as a set of characters. As with many of the other functions, one or two optional integer arguments can be used to specify a subset of the current string. These functions find the first (or last) character that is either present (or absent) from the argument set. The position of the given character, if located, is returned. If no such character exists then a value out of the range of any legal subscript is returned.

```
i = s2.find_first_of ("aeiou");           // find first vowel
j = s2.find_first_not_of ("aeiou", i);    // next non-vowel
```

12.3 An Example Function – Split a Line into Words

Obtaining the Sample Program

The split function can be found in the concordance program in file `concord.cpp`.

In this section we will illustrate the use of some of the string functions by defining a function to split a line of text into individual words. We have already made use of this function in the concordance example program in Section 9.3.3.

There are three arguments to the function. The first two are strings, describing the line of text and the separators to be used to differentiate words, respectively. The third argument is a list of strings, used to return the individual words in the line.

```
void split
   (string & text, string & separators, list<string> & words)
{
   int n = text.length();
   int start, stop;

   start = text.find_first_not_of(separators);
   while ((start >= 0) && (start < n)) {
      stop = text.find_first_of(separators, start);
      if ((stop < 0) || (stop > n)) stop = n;
      words.push_back(text.substr(start, stop - start));
      start = text.find_first_not_of(separators, stop+1);
      }
}
```

The program begins by finding the first character that is not a separator. The loop then looks for the next following character that is a separator, or uses the end of the string if no such value is found. The difference between these two is then a word, and is copied out of the text using a substring operation and inserted into the list of words. A search is then made to discover the start of the next word, and the loop continues. When the index value exceeds the limits of the string, execution stops.

Section 13. Generic Algorithms

13.1 Overview

In this section and in section 14 we will examine and illustrate each of the generic algorithms provided by the standard library. The names and a short description of each of the algorithms in this section are given in the following table. We have divided the algorithms into several categories, based on how they are typically used. This division differs from the categories used in the C++ standard definition, which is based upon which algorithms modify their arguments and which do not.

Name	*Purpose*
algorithms used to initialize a sequence Đ Section 13.2	
`fill`	fill a sequence with an initial value
`fill_n`	fill n positions with an initial value
`copy`	copy sequence into another sequence
`copy_backward`	copy sequence into another sequence
`generate`	initialize a sequence using a generator
`generate_n`	initialize n positions using a generator
`swap_ranges`	swap values from two parallel sequences
searching algorithms Đ Section 13.3	
`find`	find an element matching the argument
`find_if`	find an element satisfying a condition
`adjacent_find`	find consecutive duplicate elements
`find_first_of`	find the first occurrence of one member of a sequence in another sequence
`find_end`	find the last occurrence of a sub-sequence in a sequence
`search`	match a sub-sequence within a sequence
`max_element`	find the maximum value in a sequence
`min_element`	find the minimum value in a sequence
`mismatch`	find first mismatch in parallel sequences
in-place transformations Đ Section 13.4	
`reverse`	reverse the elements in a sequence
`replace`	replace specific values with new value
`replace_if`	replace elements matching predicate
`rotate`	rotate elements in a sequence around a point
`partition`	partition elements into two groups

Name	Purpose
stable_partition	partition preserving original ordering
next_permutation	generate permutations in sequence
prev_permutation	generate permutations in reverse sequence
inplace_merge	merge two adjacent sequences into one
random_shuffle	randomly rearrange elements in a sequence
removal algorithms Ð Section 13.5	
remove	remove elements that match condition
unique	remove all but first of duplicate values in sequences
scalar generating algorithms ÐÊSection 13.6	
count	count number of elements matching value
count_if	count elements matching predicate
accumulate	reduce sequence to a scalar value
inner_product	inner product of two parallel sequences
equal	check two sequences for equality
lexicographical_compare	compare two sequences
sequence generating algorithms ÐÊSection 13.7	
transform	transform each element
partial_sum	generate sequence of partial sums
adjacent_difference	generate sequence of adjacent differences
miscellaneous operations Ð Section 13.8	
for_each	apply a function to each element of collection

In this section we will illustrate the use of each algorithm with a series of short examples. Many of the algorithms are also used in the sample programs provided in the sections on the various container classes. These cross references have been noted where appropriate.

All of the short example programs described in this section have been collected in a number of files, named alg1.cpp through alg6.cpp. In the files, the example programs have been augmented with output statements describing the test programs and illustrating the results of executing the algorithms. In order to not confuse the reader with unnecessary detail, we have generally omitted these output statements from the descriptions here. If you wish to see the text programs complete with output statements, you can compile and execute these test files. The expected output from these programs is also included in the distribution.

13.1.1 Include Files

To use any of the generic algorithms you must first include the appropriate header file. The majority of the functions are defined in the header file `algorithm`. The functions `accumulate()`, `inner_product()`, `partial_sum()`, and `adjacent_difference()` are defined in the header file `numeric`.

```
# include <algorithm>
# include <numeric>
```

13.2 Initialization Algorithms

The first set of algorithms we will cover are those that are chiefly, although not exclusively, used to initialize a newly created sequence with certain values. The standard library provides several initialization algorithms. In our discussion we'll provide examples of how to apply these algorithms, and suggest how to choose one algorithm over another.

Obtaining the source

The sample programs described in this section can be found in the file `alg1.cpp`.

13.2.1 Fill a Sequence with An Initial Value

The `fill()` and `fill_n()` algorithms are used to initialize or reinitialize a sequence with a fixed value. Their declarations are as follows:

```
void fill (ForwardIterator first, ForwardIterator last, const T&);
void fill_n (OutputIterator, Size, const T&);
```

Different Initialization Algorithms

The initialization algorithms all overwrite every element in a container. The difference between the algorithms is the source for the values used in initialization. The `fill()` algorithm repeats a single value, the `copy()` algorithm reads values from a second container, and the `generate()` algorithm invokes a function for each new value.

The example program illustrates several uses of the algorithm:

```
void fill_example ()
   // illustrate the use of the fill algorithm
{
      // example 1, fill an array with initial values
   char buffer[100], * bufferp = buffer;
   fill (bufferp, bufferp + 100, '\0');
   fill_n (bufferp, 10, 'x');

      // example 2, use fill to initialize a list
   list<string> aList(5, "nothing");
   fill_n (inserter(aList, aList.begin()), 10, "empty");

      // example 3, use fill to overwrite values in list
   fill (aList.begin(), aList.end(), "full");

      // example 4, fill in a portion of a collection
   vector<int> iVec(10);
   generate (iVec.begin(), iVec.end(), iotaGen(1));
   vector<int>::iterator & seven =
         find(iVec.begin(), iVec.end(), 7);
   fill (iVec.begin(), seven, 0);
}
```

In example 1, an array of character values is declared. The `fill()` algorithm is invoked to initialize each location in this array with a null character value. The first 10 positions are then replaced with the character `'x'` by using the algorithm `fill_n()`. Note that the `fill()` algorithm requires both starting and past-end iterators as arguments, whereas the `fill_n()` algorithm uses a starting iterator and a count.

Example 2 illustrates how, by using an *insert iterator* (see Section 2.4), the `fill_n()` algorithm can be used to initialize a variable length container, such as a list. In this case the list initially contains five elements, all holding the text `"nothing"`. The call on `fill_n()` then inserts ten instances of the string `"empty"`. The resulting list contains fifteen elements.

The third and fourth examples illustrate how `fill()` can be used to change the values in an existing container. In the third example each of the fifteen elements in the list created in example 2 is replaced by the string `"full"`.

Example 4 overwrites only a portion of a list. Using the algorithm `generate()` and the function object ***iotaGen***, which we will describe in the next section, a vector is initialized to the values 1 2 3 ... 10. The `find()` algorithm (Section 13.3.1) is then used to locate the position of the element 7, saving the location in an iterator appropriate for the ***vector*** data type. The `fill()` call then replaces all values up to, but not including, the 7 entry with the value 0. The resulting vector has six zero fields, followed by the values 7, 8, 9 and 10.

The `fill()` and `fill_n()` algorithm can be used with all the container classes contained in the standard library, although insert iterators must be used with ordered containers, such as a ***set***.

13.2.2 Copy One Sequence Into Another Sequence

The algorithms copy() and copy_backward() are versatile functions that can be used for a number of different purposes, and are probably the most commonly executed algorithms in the standard library. The declarations for these algorithms are as follows:

```
OutputIterator copy (InputIterator first, InputIterator last,
         OutputIterator result);

BidirectionalIterator copy_backward
   (BidirectionalIterator first, BidirectionalIterator last,
    BidirectionalIterator result);
```

Appending Several Copies

The result returned by the copy() function is a pointer to the end of the copied sequence. To make a catenation of values, the result of one copy() operation can be used as a starting iterator in a subsequent copy().

Uses of the copy algorithm include:

- Duplicating an entire sequence by copying into a new sequence
- Creating sub-sequences of an existing sequence
- Adding elements into a sequence
- Copying a sequence from input or to output
- Converting a sequence from one form into another

These are illustrated in the following sample program.

```
void copy_example()
   // illustrate the use of the copy algorithm
{
   char * source = "reprise";
   char * surpass = "surpass";
   char buffer[120], * bufferp = buffer;

      // example 1, a simple copy
   copy (source, source + strlen(source) + 1, bufferp);

      // example 2, self copies
   copy (bufferp + 2, bufferp + strlen(buffer) + 1, bufferp);
   int buflen = strlen(buffer) + 1;
   copy_backward (bufferp, bufferp + buflen, bufferp + buflen + 3);
   copy (surpass, surpass + 3, bufferp);

      // example 3, copy to output
   copy (bufferp, bufferp + strlen(buffer),
            ostream_iterator<char,char>(cout));
   cout << endl;

      // example 4, use copy to convert type
   list<char> char_list;
   copy (bufferp, bufferp + strlen(buffer),
            inserter(char_list, char_list.end()));
   char * big = "big ";
   copy (big, big + 4, inserter(char_list, char_list.begin()));

   char buffer2 [120], * buffer2p = buffer2;
   * copy (char_list.begin(), char_list.end(), buffer2p) = '\0';
   cout << buffer2 << endl;
}
```

The first call on `copy(),` in example 1, simply copies the string pointed to by the variable `source` into a buffer, resulting in the buffer containing the text `"reprise".` Note that the ending position for the copy is one past the terminating null character, thus ensuring the null character is included in the copy operation.

The `copy()` operation is specifically designed to permit self-copies, i.e., copies of a sequence onto itself, as long as the destination iterator does not fall within the range formed by the source iterators. This is illustrated by example 2. Here the copy begins at position 2 of the buffer and extends to the end, copying characters into the beginning of the buffer. This results in the buffer holding the value `"prise".`

The second half of example 2 illustrates the use of the `copy_backward()` algorithm. This function performs the same task as the `copy()` algorithm, but moves elements from the end of the sequence first, progressing to the front of the sequence. (If you think of the argument as a string, characters are moved starting from the right and progressing to the left.) In this case the result will be that buffer will be assigned the value `"priprise".` The first three characters are then modified by another `copy()` operation to the values `"sur",` resulting in buffer holding the `value "surprise".`

copy_backwards

In the `copy_backwards` algorithm, note that it is the order of transfer, and not the elements themselves that is "`backwards`"; the relative placement of moved values in the target is the same as in the source.

Example 3 illustrates `copy()` being used to move values to an output stream. (See Section 2.3.2). The target in this case is an `ostream_iterator` generated for the output stream `cout`. A similar mechanism can be used for input values. For example, a simple mechanism to copy every word in the input stream into a list is the following call on `copy():`

```
list<string> words;
istream_iterator<string, char> in_stream(cin), eof;

copy(in_stream, eof, inserter(words, words.begin()));
```

This technique is used in the spell checking program described in Section 8.3.

Copy can also be used to convert from one type of stream to another. For example, the call in example 4 of the sample program copies the characters held in the buffer one by one into a list of characters. The call on `inserter()` creates an insert iterator, used to insert values into the list. The first call on `copy()` places the string `surprise`, created in example 2, into the list. The second call on `copy()` inserts the values from the string `Òbig Ò` onto the front of the list, resulting in the list containing the characters `big surprise`. The final call on `copy()` illustrates the reverse process, copying characters from a list back into a character buffer.

13.2.3 Initialize a Sequence with Generated Values

A *generator* is a function that will return a series of values on successive invocations. Probably the generator you are most familiar with is a random number generator. However, generators can be constructed for a variety of different purposes, including initializing sequences.

Like fill() and fill_n(), the algorithms generate() and generate_n() are used to initialize or reinitialize a sequence. However, instead of a fixed argument, these algorithms draw their values from a generator. The declarations of these algorithms are as follows:

```
void generate (ForwardIterator, ForwardIterator, Generator);
void generate_n (OutputIterator, Size, Generator);
```

Our example program shows several uses of the generate algorithm to initialize a sequence.

```
string generateLabel () {
   // generate a unique label string of the form L_ddd
   static int lastLabel = 0;
   char labelBuffer[80];
   ostrstream ost(labelBuffer, 80);
   ost << "L_" << lastLabel++ << '\0';
   return string(labelBuffer);
}

void generate_example ()
   // illustrate the use of the generate and generate_n algorithms
{
      // example 1, generate a list of label values
   list<string> labelList;
   generate_n (inserter(labelList, labelList.begin()),
         4, generateLabel);

      // example 2, generate an arithmetic progression
   vector<int> iVec(10);
   generate (iVec.begin(), iVec.end(), iotaGen(2));
   generate_n (iVec.begin(), 5, iotaGen(7));
   }
```

A generator can be constructed as a simple function that “remembers” information about its previous history in one or more static variables. An example is shown in the beginning of the example program, where the function generateLabel() is described. This function creates a sequence of unique string labels, such as might be needed by a compiler. Each invocation on the function generateLabel() results in a new string of the form L_ddd, each with a unique digit value. Because the variable named lastLabel is declared as static, its value is remembered from one invocation to the next. The first example of the sample program illustrates how this function might be used in combination with the generate_n() algorithm to initialize a list of four label values.

As we described in Section 3, in the Standard Library a function is any object that will respond to the function call operator. Using this fact, classes can easily be constructed as functions. The class ***iotaGen,*** which we described in Section 3.3, is an example. The ***iotaGen*** function object creates a generator for an integer arithmetic sequence. In the second example in the sample program, this sequence is used to initialize a vector with the integer values 2 through 11. A call on generate_n() is then used to overwrite the first 5 positions of the vector with the values 7 through 11, resulting in the vector 7 8 9 10 11 7 8 9 10 11.

13.2.4 Swap Values from Two Parallel Ranges

The template function swap() can be used to exchange the values of two objects of the same type. It has the following definition:

```
template <class T> void swap (T& a, T& b)
{
   T temp(a);
   a = b;
   b = temp;
}
```

The function is generalized to iterators in the function named iter_swap(). The algorithm swap_ranges() then extends this to entire sequences. The values denoted by the first sequence are exchanged with the values denoted by a second, parallel sequence. The description of the swap_ranges() algorithm is as follows:

```
ForwardIterator swap_ranges
   (ForwardIterator first, ForwardIterator last,
         ForwardIterator first2);
```

The second range is described only by a starting iterator. It is assumed (but not verified) that the second range has at least as many elements as the first range. We use both functions alone and in combination in the example program.

Parallel Sequences

A number of algorithms operate on two parallel sequences. In most cases the second sequence is identified using only a starting iterator, not a starting and ending iterator pair. It is assumed, but never verified, that the second sequence is at least as large as the first. Errors will occur if this condition is not satisfied.

```
void swap_example ()
       // illustrate the use of the algorithm swap_ranges
{
      // first make two parallel sequences
   int data[] = {12, 27, 14, 64}, *datap = data;
   vector<int> aVec(4);
   generate(aVec.begin(), aVec.end(), iotaGen(1));

      // illustrate swap and iter_swap
   swap(data[0], data[2]);
   vector<int>::iterator last = aVec.end(); last--;
   iter_swap(aVec.begin(), last);

      // now swap the entire sequence
   swap_ranges (aVec.begin(), aVec.end(), datap);
}
```

13.3 Searching Operations

The next category of algorithms we will describe are those that are used to locate elements within a sequence that satisfy certain properties. Most commonly the result of a search is then used as an argument to a further operation, such as a copy (Section 13.4.4), a partition (Section 13.2.2) or an in-place merge (Section 13.4.6.)

The searching routines described in this section return an iterator that identifies the first element that satisfies the search condition. It is common to store this value in an iterator variable, as follows:

```
list<int>::iterator where;
where = find(aList.begin(), aList.end(), 7);
```

If you want to locate *all* the elements that satisfy the search conditions you must write a loop. In that loop, the value yielded by a previous search is first advanced (since otherwise the value yielded by the previous search would once again be returned), and the resulting value is used as a starting point for the new search. For example, the following loop from the `adjacent_find()` example program (Section 13.3.2) will print the value of all repeated characters in a string argument.

Obtaining the Source

The example functions described in this section can be found in the file `alg2.cpp`.

```
while ((where = adjacent_find(where, stop)) != stop) {
   cout << "double " << *where << " in position "
      << where - start << endl;
   ++where;
   }
```

Check Search Results

The searching algorithms in the standard library all return the end-of-sequence iterator if no value is found that matches the search condition. As it is generally illegal to dereference the end-of-sequence value, it is important to check for this condition before proceeding to use the result of a search.

Many of the searching algorithms have an optional argument that can specify a function to be used to compare elements, in place of the equality operator for the container element type (operator `==`). In the descriptions of the algorithms we write these optional arguments inside a square bracket, to indicate they need not be specified if the standard equality operator is acceptable.

13.3.1 Find an Element Satisfying a Condition

There are two algorithms, `find()` and `find_if()`, that are used to find the first element that satisfies a condition. The declarations of these two algorithms are as follows:

```
InputIterator find_if (InputIterator first, InputIterator last,
      Predicate);

InputIterator find (InputIterator first, InputIterator last,
      const T&);
```

The algorithm `find_if()` takes as argument a predicate function, which can be any function that returns a boolean value (see Section 3.2). The `find_if()` algorithm returns a new iterator that designates the first element in the sequence that satisfies the predicate. The second argument, the past-the-end iterator, is returned if no element is found that matches the requirement. Because the resulting value is an iterator, the dereference operator (the `*` operator) must be used to obtain the matching value. This is illustrated in the example program.

The second form of the algorithm, `find()`, replaces the predicate function with a specific value, and returns the first element in the sequence that tests

equal to this value, using the appropriate equality operator (the == operator) for the given data type.

The following example program illustrates the use of these algorithms:

Searching Sets and Maps

These algorithms perform a linear sequential search through the associated structures. The **set** *and* **map** *data structures, which are ordered, provide their own* `find()` *member functions, which are more efficient. Because of this, the generic* `find()` *algorithm should not be used with* **set** *and* **map.**

```
void find_test ()
   // illustrate the use of the find algorithm
{
   int vintageYears[] = {1967, 1972, 1974, 1980, 1995};
   int * start = vintageYears;
   int * stop = start + 5;
   int * where = find_if (start, stop, isLeapYear);
   if (where != stop)
      cout << "first vintage leap year is " << *where << endl;
   else
      cout << "no vintage leap years" << endl;
   where = find(start, stop, 1995);
   if (where != stop)
      cout << "1995 is position " << where - start
         << " in sequence" << endl;
   else
      cout "1995 does not occur in sequence" << endl;
}
```

13.3.2 Find Consecutive Duplicate Elements

The `adjacent_find()` algorithm is used to discover the first element in a sequence equal to the next immediately following element. For example, if a sequence contained the values 1 4 2 5 6 6 7 5, the algorithm would return an iterator corresponding to the first 6 value. If no value satisfying the condition is found, then the end-of-sequence iterator is returned. The declaration of the algorithm is as follows:

```
ForwardIterator adjacent_find (ForwardIterator first,
   ForwardIterator last [, BinaryPredicate ] );
```

The first two arguments specify the sequence to be examined. The optional third argument must be a binary predicate (a binary function returning a boolean value). If present, the binary function is used to test adjacent elements, otherwise the equality operator (operator ==) is used.

The example program searches a text string for adjacent letters. In the example text these are found in positions 5, 7, 9, 21 and 37. The increment is necessary inside the loop in order to avoid the same position being discovered repeatedly.

```
void adjacent_find_example ()
   // illustrate the use of the adjacent_find instruction
{
   char * text = "The bookkeeper carefully opened the door.";
   char * start = text;
   char * stop = text + strlen(text);
   char * where = start;

   cout << "In the text: " << text << endl;
   while ((where = adjacent_find(where, stop)) != stop) {
      cout << "double " << *where
         << " in position " << where - start << endl;
```

```
      ++where;
   }
}
```

13.3.3 Find the first occurrence of any value from a sequence

The algorithm `find_first_of()` is used to find the first occurrence of some element from one sequence that is also contained in another sequence.

```
ForwardIterator1 find_first_of
        (ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2
         [, BinaryPredicate pred ] );
```

The algorithm returns a new iterator that designates the first element found in `[first1,last1)` that is also contained in `[first2,last2)`. If no match is found then the second argument is returned . Because the resulting value is an iterator, the dereference operator (the `*` operator) must be used to obtain the matching value. This is illustrated in the example program.

The following example program illustrates the use of this algorithm:

```
void find_test ()
   // illustrate the use of the find algorithm
{
   int vintageYears[] = {1967, 1972, 1974, 1980, 1995};
   int requestedYears[] = [1923, 1970, 1980, 1974 };
   int * start = vintageYears;
   int * stop = start + 5;
   int * where = find_first_of (start, stop,
                           requestedyears,requestedyears+4 );
   if (where != stop)
      cout << "first requested vintage year is " << *where << endl;
   else
      cout << "no requested vintage years" << endl;
}

// The output would indicate 1974.
```

Searching Sets and Maps

The **basic_string** *class provides its own versions of the* **find_first_of** *and* **find_end** *algorithms, including several convience overloads of the basic pattern indicated here.*

Note that this algorithm, unlike many that manipulate two sequences, uses a starting and ending iterator pair for both sequences, not just the first sequence.

Like the algorithms `equal()` and `mismatch()`, an alternative version of `find_first_of()` takes an optional binary predicate that is used to compare elements from the two sequences.

13.3.4 Find a Sub-sequence within a Sequence

The algorithms `search()` and `search_n()` are used to locate the beginning of a particular sub-sequence within a larger sequence. The easiest example to understand is the problem of looking for a particular substring within a larger string, although the algorithm can be generalized to other uses. The arguments are assumed to have at least the capabilities of forward iterators.

```
ForwardIterator search
   (ForwardIterator first1, ForwardIterator last1,
   ForwardIterator first2, ForwardIterator last2
   [, BinaryPredicate ]);
```

Speed of Search

In the worst case, the number of comparisons performed by the algorithm `search()` is the product of the number of elements in the two sequences. Except in rare cases, however, this worst case behavior is highly unlikely.

Suppose, for example, that we wish to discover the location of the string `"ration"` in the string `"dreams and aspirations".` The solution to this problem is shown in the example program. If no appropriate match is found, the value returned is the past-the-end iterator for the first sequence.

```
void search_example ()
      // illustrate the use of the search algorithm
{
   char * base = "dreams and aspirations";
   char * text = "ration";

   char * where = search(base, base + strlen(base),
         text, text + strlen(text));

   if (*where != '\0')
      cout << "substring position: " << where - base << endl;
   else
      cout << "substring does not occur in text" << endl;
}
```

Note that this algorithm, unlike many that manipulate two sequences, uses a starting and ending iterator pair for both sequences, not just the first sequence.

Like the algorithms `equal()` and `mismatch(),` an alternative version of `search()` takes an optional binary predicate that is used to compare elements from the two sequences.

13.3.5 Find the last occurrence of a Sub-sequence

The algorithm `find_end()` is used to locate the beginning of a the last occurrence of a particular sub-sequence within a larger sequence. The easiest example to understand is the problem of looking for a particular substring within a larger string, although the algorithm can be generalized to other uses. The arguments are assumed to have at least the capabilities of forward iterators.

```
ForwardIterator find_end
   (ForwardIterator first1, ForwardIterator last1,
   ForwardIterator first2, ForwardIterator last2
   [, BinaryPredicate ]);
```

Suppose, for example, that we wish to discover the location of the last occurrence of the string "le" in the string "The road less traveled". The solution to this problem is shown in the example program. If no appropriate match is found, the value returned is the past-the-end iterator for the first sequence.

Speed of Find_end

As with search, n the worst case, the number of comparisons performed by the algorithm find_end() is the product of the number of elements in the two sequences.

```
void find_end_example ()
      // illustrate the use of the find_end algorithm
{
   char * base = "The road less traveled";
   char * text = "le";

   char * where = find(base, base + strlen(base),
         text, text + strlen(text));

   if (*where != '\0')
      cout << "substring position: " << where - base << endl;
   else
      cout << "substring does not occur in text" << endl;
}
```

Note that this algorithm, unlike many that manipulate two sequences, uses a starting and ending iterator pair for both sequences, not just the first sequence.

Like the algorithms find_first_of() and search(), an alternative version of find_end() takes an optional binary predicate that is used to compare elements from the two sequences.

13.3.6 Locate Maximum or Minimum Element

The functions max() and min() can be used to find the maximum and minimum of a pair of values. These can optionally take a third argument that defines the comparison function to use in place of the less-than operator (operator <). The arguments are values, not iterators:

```
template <class T>
   const T& max(const T& a, const T& b [, Compare ] );
template <class T>
   const T& min(const T& a, const T& b [, Compare ] );
```

The maximum and minimum functions are generalized to entire sequences by the generic algorithms max_element() and min_element(). For these functions the arguments are input iterators.

```
ForwardIterator max_element (ForwardIterator first,
      ForwardIterator last [, Compare ] );
ForwardIterator min_element (ForwardIterator first,
      ForwardIterator last [, Compare ] );
```

Largest and Smallest Elements of a Set

The maximum and minimum algorithms can be used with all the data types provided by the standard library. However, for the ordered data types, **set** *and* **map**, *the maximum or minimum values are more easily accessed as the first or last elements in the structure.*

These algorithms return an iterator that denotes the largest or smallest of the values in a sequence, respectively. Should more than one value satisfy the requirement, the result yielded is the first satisfactory value. Both algorithms can optionally take a third argument, which is the function to be used as the comparison operator in place of the default operator.

The example program illustrates several uses of these algorithms. The function named `split()` used to divide a string into words in the string example is described in Section 12.3. The function `randomInteger()` is described in Section 2.2.5.

```
void max_min_example ()
   // illustrate use of max_element and min_element algorithms
{
   // make a vector of random numbers between 0 and 99
   vector<int> numbers(25);
   for (int i = 0; i < 25; i++)
      numbers[i] = randomInteger(100);

   // print the maximum
   vector<int>::iterator max =
      max_element(numbers.begin(), numbers.end());
   cout << "largest value was " << * max << endl;

   // example using strings
   string text =
      "It was the best of times, it was the worst of times.";
   list<string> words;
   split (text, " .,!:;", words);
   cout << "The smallest word is "
        << * min_element(words.begin(), words.end())
        << " and the largest word is "
        << * max_element(words.begin(), words.end())
        << endl;
}
```

13.3.7 Locate the First Mismatched Elements in Parallel Sequences

The name `mismatch()` might lead you to think this algorithm was the inverse of the `equal()` algorithm, which determines if two sequences are equal (see Section 13.6.4). Instead, the `mismatch()` algorithm returns a ***pair*** of iterators that together indicate the first positions where two parallel sequences have differing elements. (The structure ***pair*** is described in Section 9.1). The second sequence is denoted only by a starting position, without an ending position. It is assumed (but not checked) that the second sequence contains at least as many elements as the first. The arguments and return type for `mismatch()` can be described as follows:

```
pair<InputIterator, InputIterator> mismatch
   (InputIterator first1, InputIterator last1,
      InputIterator first2 [, BinaryPredicate ] );
```

The elements of the two sequences are examined in parallel, element by element. When a mismatch is found, that is, a point where the two sequences differ, then a ***pair*** containing iterators denoting the locations of the

two differing elements is constructed and returned. If the first sequence becomes exhausted before discovering any mismatched elements, then the resulting pair contains the ending value for the first sequence, and the last value examined in the second sequence. (The second sequence need not yet be exhausted).

The example program illustrates the use of this procedure. The function `mismatch_test()` takes as arguments two string values. These are lexicographically compared and a message printed indicating their relative ordering. (This is similar to the analysis performed by the `lexicographic_compare()` algorithm, although that function simply returns a boolean value.) Because the `mismatch()` algorithm assumes the second sequence is at least as long as the first, a comparison of the two string lengths is performed first, and the arguments are reversed if the second string is shorter than the first. After the call on `mismatch()` the elements of the resulting pair are separated into their component parts. These parts are then tested to determine the appropriate ordering.

```
void mismatch_test (char * a, char * b)
      // illustrate the use of the mismatch algorithm
{
   pair<char *, char *> differPositions(0, 0);
   char * aDiffPosition;
   char * bDiffPosition;

   if (strlen(a) < strlen(b)) {
      // make sure longer string is second
      differPositions = mismatch(a, a + strlen(a), b);
      aDiffPosition = differPositions.first;
      bDiffPosition = differPositions.second;
      }
   else {
      differPositions = mismatch(b, b + strlen(b), a);
      // note following reverse ordering
      aDiffPosition = differPositions.second;
      bDiffPosition = differPositions.first;
      }

   // compare resulting values
   cout << "string " << a;
   if (*aDiffPosition == *bDiffPosition)
      cout << " is equal to ";
   else if (*aDiffPosition < *bDiffPosition)
      cout << " is less than ";
   else
      cout << " is greater than ";
   cout << b << endl;
}
```

A second form of the `mismatch()` algorithm is similar to the one illustrated, except it accepts a binary predicate as a fourth argument. This binary function is used to compare elements, in place of the `==` operator.

13.4 In-Place Transformations

Obtaining the Source

The example functions described in this section can be found in the file `alg3.cpp`.

The next category of algorithms in the standard library that we examine are those used to modify and transform sequences without moving them from their original storage locations. A few of these routines, such as `replace()`, include a *copy* version as well as the original in-place transformation algorithms. For the others, should it be necessary to preserve the original, a copy of the sequence should be created before the transformations are applied. For example, the following illustrates how one can place the reversal of one vector into another newly allocated vector.

```
vector<int> newVec(aVec.size());
copy (aVec.begin(), aVec.end(), newVec.begin()); // first copy
reverse (newVec.begin(), newVec.end());     // then reverse
```

Many of the algorithms described as sequence generating operations, such as `transform()` (Section 13.7.1), or `partial_sum()` (Section 13.7.2), can also be used to modify a value in place by simply using the same iterator as both input and output specification.

13.4.1 Reverse Elements in a Sequence

The algorithm `reverse()` reverses the elements in a sequence, so that the last element becomes the new first, and the first element the new last. The arguments are assumed to be bidirectional iterators, and no value is returned.

```
void reverse (BidirectionalIterator first,
   BidirectionalIterator last);
```

The example program illustrates two uses of this algorithm. In the first, an array of characters values is reversed. The algorithm `reverse()` can also be used with list values, as shown in the second example. In this example, a list is initialized with the values 2 to 11 in increasing order. (This is accomplished using the ***iotaGen*** function object introduced in Section 3.3). The list is then reversed, which results in the list holding the values 11 to 2 in decreasing order. Note, however, that the list data structure also provides its own `reverse()` member function.

```
void reverse_example ()
   // illustrate the use of the reverse algorithm
{
   // example 1, reversing a string
char * text = "Rats live on no evil star";
reverse (text, text + strlen(text));
cout << text << endl;

  // example 2, reversing a list
list<int> iList;
generate_n (inserter(iList, iList.begin()), 10, iotaGen(2));
reverse (iList.begin(), iList.end());
}
```

13.4.2 Replace Certain Elements With Fixed Value

The algorithms replace() and replace_if() are used to replace occurrences of certain elements with a new value. In both cases the new value is the same, no matter how many replacements are performed. Using the algorithm replace(), all occurrences of a particular test value are replaced with the new value. In the case of replace_if(), all elements that satisfy a predicate function are replaced by a new value. The iterator arguments must be forward iterators.

The algorithms replace_copy() and replace_copy_if() are similar to replace() and replace_if(), however they leave the original sequence intact and place the revised values into a new sequence, which may be a different type.

```
void replace (ForwardIterator first, ForwardIterator last,
         const T&, const T&);
void replace_if (ForwardIterator first, ForwardIterator last,
         Predicate, const T&);
OutputIterator replace_copy (InputIterator, InputIterator,
         OutputIterator, const T&, const T&);
OutputIterator replace_copy (InputIterator, InputIterator,
         OutputIterator, Predicate, const T&);
```

In the example program, a vector is initially assigned the values 0 1 2 3 4 5 4 3 2 1 0. A call on replace() replaces the value 3 with the value 7, resulting in the vector 0 1 2 7 4 5 4 7 2 1 0. The invocation of replace_if() replaces all even numbers with the value 9, resulting in the vector 9 1 9 7 9 5 9 7 9 1 9.

```
void replace_example ()
       // illustrate the use of the replace algorithm
{
      // make vector 0 1 2 3 4 5 4 3 2 1 0
   vector<int> numbers(11);
   for (int i = 0; i < 11; i++)
      numbers[i] = i < 5 ? i : 10 - i;

      // replace 3 by 7
   replace (numbers.begin(), numbers.end(), 3, 7);

      // replace even numbers by 9
   replace_if (numbers.begin(), numbers.end(), isEven, 9);

      // illustrate copy versions of replace
   int aList[] = {2, 1, 4, 3, 2, 5};
   int bList[6], cList[6], j;
   replace_copy (aList, aList+6, &bList[0], 2, 7);
   replace_copy_if (bList, bList+6, &cList[0],
         bind2nd(greater<int>(), 3), 8);
}
```

The example program also illustrates the use of the replace_copy algorithms. First, an array containing the values 2 1 4 3 2 5 is created. This is modified by replacing the 2 values with 7, resulting in the array 7 1 4 3 7 5. Next, all values larger than 3 are replaced with the value 8, resulting in the array values 8 1 8 3 8 8. In the latter case the bind2nd() adaptor is used, to

modify the binary greater-than function by binding the 2nd argument to the constant value 3, thereby creating the unary function `x > 3`.

13.4.3 Rotate Elements Around a Midpoint

A rotation of a sequence divides the sequence into two sections, then swaps the order of the sections, maintaining the relative ordering of the elements within the two sections. Suppose, for example, that we have the values 1 to 10 in sequence.

1 2 3 4 5 6 7 8 9 10

If we were to rotate around the element 7, the values 7 to 10 would be moved to the beginning, while the elements 1 to 6 would be moved to the end. This would result in the following sequence.

7 8 9 10 1 2 3 4 5 6

When you invoke the algorithm `rotate()`, the starting point, midpoint, and past-the-end location are all denoted by forward iterators:

```
void rotate (ForwardIterator first, ForwardIterator middle,
   ForwardIterator last);
```

The prefix portion, the set of elements following the start and not including the midpoint, is swapped with the suffix, the set of elements between the midpoint and the past-the-end location. Note, as in the illustration presented earlier, that these two segments need not be the same length.

```
void rotate_example()
       // illustrate the use of the rotate algorithm
{
       // create the list 1 2 3 ... 10
   list<int> iList;
   generate_n(inserter(iList, iList.begin()), 10, iotaGen(1));

      // find the location of the seven
   list<int>::iterator & middle =
         find(iList.begin(), iList.end(), 7);

      // now rotate around that location
   rotate (iList.begin(), middle, iList.end());

      // rotate again around the same location
   list<int> cList;
   rotate_copy (iList.begin(), middle, iList.end(),
      inserter(cList, cList.begin()));
}
```

The example program first creates a list of the integers in order from 1 to 10. Next, the `find()` algorithm (Section 13.3.1) is used to find the location of the element 7. This is used as the midpoint for the rotation.

A second form of `rotate()` copies the elements into a new sequence, rather than rotating the values in place. This is also shown in the example program, which once again rotates around the middle position (now containing a 3).

The resulting list is 3 4 5 6 7 8 9 10 1 2. The values held in `iList` remain unchanged.

13.4.4 Partition a Sequence into Two Groups

A *partition* is formed by moving all the elements that satisfy a predicate to one end of a sequence, and all the elements that fail to satisfy the predicate to the other end. Partitioning elements is a fundamental step in certain sorting algorithms, such as "quicksort."

```
BidirectionalIterator partition
   (BidirectionalIterator, BidirectionalIterator, Predicate);

BidirectionalIterator stable_partition
   (BidirectionalIterator, BidirectionalIterator, Predicate);
```

There are two forms of partition supported in the standard library. The first, provided by the algorithm `partition()`, guarantees only that the elements will be divided into two groups. The result value is an iterator that describes the final midpoint between the two groups; it is one past the end of the first group.

In the example program the initial vector contains the values 1 to 10 in order. The partition moves the even elements to the front, and the odd elements to the end. This results in the vector holding the values 10 2 8 4 6 5 7 3 9 1, and the midpoint iterator pointing to the element 5.

```
void partition_example ()
       // illustrate the use of the partition algorithm
{
       // first make the vector 1 2 3 ... 10
   vector<int> numbers(10);
   generate(numbers.begin(), numbers.end(), iotaGen(1));

      // now put the even values low, odd high
   vector<int>::iterator result =
      partition(numbers.begin(), numbers.end(), isEven);
   cout << "middle location " << result - numbers.begin() << endl;

      // now do a stable partition
   generate (numbers.begin(), numbers.end(), iotaGen(1));
   stable_partition (numbers.begin(), numbers.end(), isEven);
}
```

The relative order of the elements within a partition in the resulting vector may not be the same as the values in the original vector. For example, the value 4 preceded the element 8 in the original, yet in the result it may follow the element 8. A second version of partition, named `stable_partition()`, guarantees the ordering of the resulting values. For the sample input shown in the example, the stable partition would result in the sequence 2 4 6 8 10 1 3 5 7 9. The `stable_partition()` algorithm is slightly slower and uses more memory than the `partition()` algorithm, so when the order of elements is not important you should use `partition()`.

Partitions

While there is a unique `stable_partition()` for any sequence, the `partition()` algorithm can return any number of values. The following, for example, are all legal partitions of the example problem.

2 4 6 8 10 1 3 5 7 9

10 8 6 4 2 5 7 9 3 1

2 6 4 8 10 3 5 7 9 1

6 4 2 10 8 5 3 7 9 1.

13.4.5 Generate Permutations in Sequence

Ordering Permutations

Permutations can be ordered, with the smallest permutation being the one in which values are listed smallest to largest, and the largest being the sequence that lists values largest to smallest. Consider, for example, the permutations of the integers 1 2 3. The six permutations of these values are, in order:

1 2 3

1 3 2

2 1 3

2 3 1

3 1 2

3 2 1

Notice that in the first permutation the values are all ascending, while in the last permutation they are all descending.

A permutation is a rearrangement of values. If values can be compared against each other (such as integers, characters, or words) then it is possible to systematically construct all permutations of a sequence. There are 2 permutations of two values, for example, and six permutations of three values, and 24 permutations of four values.

The permutation generating algorithms have the following definition:

```
bool next_permutation (BidirectionalIterator first,
      BidirectionalIterator last, [ Compare ] );

bool prev_permutation (BidirectionalIterator first,
      BidirectionalIterator last, [ Compare ] );
```

The second example in the sample program illustrates the same idea, only using pointers to character arrays instead of integers. In this case a different comparison function must be supplied, since the default operator would simply compare pointer addresses.

```
bool nameCompare (char * a, char * b) { return strcmp(a, b) <= 0; }

void permutation_example ()
   // illustrate the use of the next_permutation algorithm
{
      // example 1, permute the values 1 2 3
   int start [] = { 1, 2, 3};
   do
      copy (start, start + 3,
            ostream_iterator<int,char> (cout, " ")), cout << endl;
   while (next_permutation(start, start + 3));

      // example 2, permute words
   char * words = {"Alpha", "Beta", "Gamma"};
   do
      copy (words, words + 3,
         ostream_iterator<char *,char> (cout, " ")), cout << endl;
   while (next_permutation(words, words + 3, nameCompare));

      // example 3, permute characters backwards
   char * word = "bela";
   do
      cout << word << ' ';
   while (prev_permutation (word, &word[4]));
   cout << endl;
}
```

Example 3 in the sample program illustrates the use of the reverse permutation algorithm, which generates values in reverse sequence. This example also begins in the middle of a sequence, rather than at the beginning. The remaining permutations of the word “bela,” are `beal`, `bale`, `bael`, `aleb`, `albe`, `aelb`, `aebl`, `able`, and finally, `abel`.

13.4.6 Merge Two Adjacent Sequences into One

A *merge* takes two ordered sequences and combines them into a single ordered sequence, interleaving elements from each collection as necessary to generate the new list. The `inplace_merge()` algorithm assumes a sequence is divided into two adjacent sections, each of which is ordered. The merge combines the two sections into one, moving elements as necessary. (The alternative `merge()` algorithm, described elsewhere, can be used to merge two separate sequences into one.) The arguments to `inplace_merge()` must be bidirectional iterators.

```
void inplace_merge (BidirectionalIterator first,
   BidirectionalIterator middle,
   BidirectionalIterator last [, BinaryFunction ] );
```

The example program illustrates the use of the `inplace_merge()` algorithm with a vector and with a list. The sequence 0 2 4 6 8 1 3 5 7 9 is placed into a vector. A `find()` call (Section 13.3.1) is used to locate the beginning of the odd number sequence. The two calls on `inplace_merge()` then combine the two sequences into one.

```
void inplace_merge_example ()
      // illustrate the use of the inplace_merge algorithm
{
      // first generate the sequence 0 2 4 6 8 1 3 5 7 9
   vector<int> numbers(10);
   for (int i = 0; i < 10; i++)
      numbers[i] = i < 5 ? 2 * i : 2 * (i - 5) + 1;

      // then find the middle location
   vector<int>::iterator midvec =
      find(numbers.begin(), numbers.end(), 1);

      // copy them into a list
   list<int> numList;
   copy (numbers.begin(), numbers.end(),
         inserter (numList, numList.begin()));
   list<int>::iterator midList =
         find(numList.begin(), numList.end, 1);

      // now merge the lists into one
   inplace_merge (numbers.begin(), midvec, numbers.end());
   inplace_merge (numList.begin(), midList, numList.end());
}
```

13.4.7 Randomly Rearrange Elements in a Sequence

The algorithm `random_shuffle()` randomly rearranges the elements in a sequence. Exactly *n* swaps are performed, where *n* represents the number of elements in the sequence. The results are, of course, unpredictable. Because the arguments must be random access iterators, this algorithm can only be used with vectors, deques, or ordinary pointers. It cannot be used with lists, sets, or maps.

```
void random_shuffle (RandomAccessIterator first,
   RandomAccessIterator last [, Generator ] );
```

An alternative version of the algorithm uses the optional third argument. This value must be a random number generator. This generator must take as an argument a positive value `m` and return a value between `0` and `m-1`. As with the `generate()` algorithm, this random number function can be any type of object that will respond to the function invocation operator.

```
void random_shuffle_example ()
   // illustrate the use of the random_shuffle algorithm
{
   // first make the vector containing 1 2 3 ... 10
   vector<int> numbers;
   generate(numbers.begin(), numbers.end(), iotaGen(1));

   // then randomly shuffle the elements
   random_shuffle (numbers.begin(), numbers.end());

   // do it again, with explicit random number generator
   struct RandomInteger {
   {
      operator() (int m) { return rand() % m; }
   } random;

   random_shuffle (numbers.begin(), numbers.end(), random);
}
```

13.5 Removal Algorithms

The following two algorithms can be somewhat confusing the first time they are encountered. Both claim to remove certain values from a sequence. But, in fact, neither one reduces the size of the sequence. Both operate by moving the values that are to be *retained* to the front of the sequence, and returning an iterator that describes where this sequence ends. Elements after this iterator are simply the original sequence values, left unchanged. This is necessary because the generic algorithm has no knowledge of the container it is working on. It only has a generic iterator. This is part of the price we pay for generic algorithms. In most cases the user will want to use this iterator result as an argument to the `erase()` member function for the container, removing the values from the iterator to the end of the sequence.

Let us illustrate this with a simple example. Suppose we want to remove the even numbers from the sequence 1 2 3 4 5 6 7 8 9 10, something we could do with the `remove_if()` algorithm. The algorithm `remove_if()` would leave us with the following sequence:

1 3 5 7 9 | 6 7 8 9 10

The vertical bar here represents the position of the iterator returned by the `remove_if()` algorithm. Notice that the five elements before the bar represent the result we want, while the five values after the bar are simply the original contents of those locations. Using this iterator value along with the end-of-sequence iterator as arguments to `erase()`, we can eliminate the unwanted values, and obtained the desired result.

Both the algorithms described here have an alternative *copy* version. The copy version of the algorithms leaves the original unchanged, and places the preserved elements into an output sequence.

What is a Name?

The algorithms in this section set up a sequence so that the desired elements are moved to the front. The remaining values are not actually removed, but the starting location for these values is returned, making it possible to remove these values by means of a subsequent call on `erase()`. Remember, *the remove algorithms do not actually remove the unwanted elements.*

13.5.1 Remove Unwanted Elements

The algorithm `remove()` eliminates unwanted values from a sequence. As with the `find()` algorithm, these can either be values that match a specific constant, or values that satisfy a given predicate. The declaration of the argument types is as follows:

```
ForwardIterator remove
   (ForwardIterator first, ForwardIterator last, const T &);
ForwardIterator remove_if
   (ForwardIterator first, ForwardIterator last, Predicate);
```

The algorithm `remove()` copies values to the front of the sequence, overwriting the location of the removed elements. All elements not removed remain in their relative order. Once all values have been examined, the remainder of the sequence is left unchanged. The iterator returned as the result of the operation provides the end of the new sequence. For example, eliminating the element 2 from the sequence 1 2 4 3 2 results in the sequence

Obtaining the Source

The example functions described in this section can be found in the file `alg4.cpp`.

1 4 3 3 2, with the iterator returned as the result pointing at the second 3. This value can be used as argument to erase() in order to eliminate the remaining elements (the 3 and the 2), as illustrated in the example program.

A copy version of the algorithms copies values to an output sequence, rather than making transformations in place.

```
OutputIterator remove_copy
         (InputIterator first, InputIterator last,
         OutputIterator result, const T &);

OutputIterator remove_copy_if
         (InputIterator first, InputIterator last,
         OutputIterator result, Predicate);
```

The use of remove() is shown in the following program.

```
void remove_example ()
   // illustrate the use of the remove algorithm
{
   // create a list of numbers
   int data[] = {1, 2, 4, 3, 1, 4, 2};
   list<int> aList;
   copy (data, data+7, inserter(aList, aList.begin()));

      // remove 2's, copy into new list
   list<int> newList;
   remove_copy (aList.begin(), aList.end(),
      back_inserter(newList), 2);

      // remove 2's in place
   list<int>::iterator where;
   where = remove (aList.begin(), aList.end(), 2);
   aList.erase(where, aList.end());

      // remove all even values
   where = remove_if (aList.begin(), aList.end(), isEven);
   aList.erase(where, aList.end());
}
```

13.5.2 Remove Runs of Similar Values

The algorithm unique() moves through a linear sequence, eliminating all but the first element from every consecutive group of equal elements. The argument sequence is described by forward iterators.

```
ForwardIterator unique (ForwardIterator first,
   ForwardIterator last [, BinaryPredicate ] );
```

As the algorithm moves through the collection, elements are moved to the front of the sequence, overwriting the existing elements. Once all unique values have been identified, the remainder of the sequence is left unchanged.

For example, a sequence such as 1 3 3 2 2 2 4 will be changed into 1 3 2 4 | 2 2 4. We have used a vertical bar to indicate the location returned by the iterator result value. This location marks the end of the unique sequence, and the beginning of the left-over elements. With most containers the value returned by the algorithm can be used as an argument in a subsequent call

on erase() to remove the undesired elements from the collection. This is illustrated in the example program.

A copy version of the algorithm moves the unique values to an output iterator, rather than making modifications in place. In transforming a list or multiset, an insert iterator can be used to change the copy operations of the output iterator into insertions.

```
OutputIterator unique_copy
      (InputIterator first, InputIterator last,
       OutputIterator result [, BinaryPredicate ] );
```

These are illustrated in the sample program:

```
void unique_example ()
   // illustrate use of the unique algorithm
{
      // first make a list of values
   int data[] = {1, 3, 3, 2, 2, 4};
   list<int> aList;
   set<int> aSet;
   copy (data, data+6, inserter(aList, aList.begin()));

      // copy unique elements into a set
   unique_copy (aList.begin(), aList.end(),
      inserter(aSet, aSet.begin()));

      // copy unique elements in place
   list<int>::iterator where;
   where = unique(aList.begin(), aList.end());

      // remove trailing values
   aList.erase(where, aList.end());
}
```

13.6 Scalar-Producing Algorithms

The next category of algorithms are those that reduce an entire sequence to a single scalar value.

Remember that two of these algorithms, accumulate() and inner_product(), are declared in the numeric header file, not the algorithm header file as are the other generic algorithms.

Obtaining the Source

The example functions described in this section can be found in the file alg5.cpp.

13.6.1 Count the Number of Elements that Satisfy a Condition

The algorithms count() and count_if() are used to discover the number of elements that match a given value or that satisfy a given predicate, respectively. Each algorithm comes in two flavors. The newer form returns the number of matches found, while the older one takes as argument a reference to a counting value (typically an integer), and increments this value (in this case he count() function itself yields no value).

The newer form of these functions is the one currently mandated by the standard. The older form is retained for two reasons: First, for backward compatibility, since older versions of the standard contained it, and second,

because the newer form requires that a compiler support partial specialization, which as of this writing is a rare thing indeed.

```
iterator_traits<InputIterator>::distance_type
count (InputIterator first, InputIterator last, const T& value);

iterator_traits<InputIterator>::distance_type
count_if (InputIterator first, InputIterator last, Predicate pred);

void count (InputIterator first, InputIterator last,
            const T&, Size &);
void count_if (InputIterator first, InputIterator last,
         Predicate, Size &);
```

The example code fragment illustrates the use of the older form of these algorithms. The call on `count()` will count the number of occurrences of the letter `e` in a sample string, while the invocation of `count_if()` will count the number of vowels.

The Resulting Count

Note that if your compiler does not support partial specialization then you will not have the versions of the `count()` algorithms that return the sum as a function result, but instead only the versions that add to the last argument in their parameter list, which is passed by reference. This means successive calls on these functions can be used to produce a cumulative sum. This also means that you must initialize the variable passed to this last argument location prior to calling one of these algorithms.

```
void count_example ()
      // illustrate the use of the count algorithm
{
   int eCount = 0;
   int vowelCount = 0;
   char * text = "Now is the time to begin";
   count (text, text + strlen(text), 'e', eCount);
   count_if (text, text + strlen(text), isVowel, vowelCount);
   cout << "There are " << eCount << " letter e's " << endl
        << "and " << vowelCount << " vowels in the text:"
        << text << endl;
}
```

13.6.2 Reduce Sequence to a Single Value

The result generated by the `accumulate()` algorithm is the value produced by placing a binary operator between each element of a sequence, and evaluating the result. By default the operator is the addition operator, `+`, however this can be replaced by any binary function. An initial value (an identity) must be provided. This value is returned for empty sequences, and is otherwise used as the left argument for the first calculation.

```
ContainerType accumulate (InputIterator first, InputIterator last,
         ContainerType initial [, BinaryFunction ] );
```

The example program illustrates the use of `accumulate()` to produce the sum and product of a vector of integer values. In the first case the identity is zero, and the default operator `+` is used. In the second invocation the identity is 1, and the multiplication operator (named ***times***) is explicitly passed as the fourth argument.

```
void accumulate_example ()
// illustrate the use of the accumulate algorithm
{
   int numbers[] = {1, 2, 3, 4, 5};
// first example, simple accumulation
   int sum = accumulate (numbers, numbers + 5, 0);
   int product =
```

```
         accumulate (numbers, numbers + 5, 1, times<int>());
   cout << "The sum of the first five integers is " << sum << endl;
   cout << "The product is " << product << endl;
// second example, with different types for initial value
   list<int> nums;
   nums = accumulate (numbers, numbers+5, nums, intReplicate);
}
list<int>& intReplicate (list<int>& nums, int n)
      // add sequence n to 1 to end of list
{
   while (n) nums.push_back(n--);
   return nums;
}
```

Neither the identity value nor the result of the binary function are required to match the container type. This is illustrated in the example program by the invocation of `accumulate()` shown in the second example above. Here the identity is an empty list. The function (shown after the example program) takes as argument a list and an integer value, and repeatedly inserts values into the list. The values inserted represent a decreasing sequence from the argument down to 1. For the example input (the same vector as in the first example), the resulting list contains the 15 values 1 2 1 3 2 1 4 3 2 1 5 4 3 2 1.

13.6.3 Generalized Inner Product

Assume we have two sequences of *n* elements each; *a1*, *a2*, ... *an* and *b1*, *b2*, ... *bn*. The *inner product* of the sequences is the sum of the parallel products, that is the value *a1* * *b1* + *a2* * *b2* + ... + *an* * *bn*. Inner products occur in a number of scientific calculations. For example, the inner product of a row times a column is the heart of the traditional matrix multiplication algorithm. A generalized inner product uses the same structure, but permits the addition and multiplication operators to be replaced by arbitrary binary functions. The standard library includes the following algorithm for computing an inner product:

```
ContainerType inner_product
   (InputIterator first1, InputIterator last1,
   InputIterator first2, ContainerType initialValue
      [ , BinaryFunction add, BinaryFunction times ] );
```

The first three arguments to the `inner_product()` algorithm define the two input sequences. The second sequence is specified only by the beginning iterator, and is assumed to contain at least as many elements as the first sequence. The next argument is an initial value, or identity, used for the summation operator. This is similar to the identity used in the `accumulate()` algorithm. In the generalized inner product function the last two arguments are the binary functions that are used in place of the addition operator, and in place of the multiplication operator, respectively.

In the example program the second invocation illustrates the use of alternative functions as arguments. The multiplication is replaced by an

equality test, while the addition is replaced by a logical *or*. The result is true if any of the pairs are equal, and false otherwise. Using an *and* in place of the *or* would have resulted in a test which was true only if *all* pairs were equal; in effect the same as the `equal()` algorithm described in the next section.

```
void inner_product_example ()
      // illustrate the use of the inner_product algorithm
{
   int a[] = {4, 3, -2};
   int b[] = {7, 3, 2};

      // example 1, a simple inner product
   int in1 = inner_product(a, a+3, b, 0);
   cout << "Inner product is " << in1 << endl;

         // example 2, user defined operations
   bool anyequal = inner_product(a, a+3, b, true,
         logical_or<bool>(), equal_to<int>());
   cout << "any equal? " << anyequal << endl;
}
```

13.6.4 Test Two Sequences for Pairwise Equality

The `equal()` algorithm tests two sequences for pairwise equality. By using an alternative binary predicate, it can also be used for a wide variety of other pair-wise tests of parallel sequences. The arguments are simple input iterators:

```
bool equal (InputIterator first, InputIterator last,
         InputIterator first2 [, BinaryPredicate] );
```

The equal() algorithm assumes, but does not verify, that the second sequence contains at least as many elements as the first. A true result is generated if all values test equal to their corresponding element. The alternative version of the algorithm substitutes an arbitrary boolean function for the equality test, and returns true if all pair-wise elements satisfy the predicate. In the sample program this is illustrated by replacing the predicate with the greater_equal() function, and in this fashion true will be returned only if all values in the first sequence are greater than or equal to their corresponding value in the second sequence.

```
void equal_example ()
   // illustrate the use of the equal algorithm
{
   int a[] = {4, 5, 3};
   int b[] = {4, 3, 3};
   int c[] = {4, 5, 3};

   cout << "a = b is: " << equal(a, a+3, b) << endl;
   cout << "a = c is: " << equal(a, a+3, c) << endl;
   cout << "a pair-wise greater-equal b is: "
      << equal(a, a+3, b, greater_equal<int>()) << endl;
}
```

Equal and Mismatch

By substituting another function for the binary predicate, the equal and mismatch algorithms can be put to a variety of different uses. Use the equal() *algorithm if you want a pairwise test that returns a* boolean *result. Use the* mismatch() *algorithm if you want to discover the* location *of elements that fail the test.*

13.6.5 Lexical Comparison

A lexical comparison of two sequences can be described by noting the features of the most common example, namely the comparison of two words for the purposes of placing them in "dictionary order." When comparing two words, the elements (that is, the characters) of the two sequences are compared in a pair-wise fashion. As long as they match, the algorithm advances to the next character. If two corresponding characters fail to match, the earlier character determines the smaller word. So, for example, everybody is smaller than everything, since the b in the former word alphabetically precedes the t in the latter word. Should one or the other sequence terminate before the other, than the terminated sequence is considered to be smaller than the other. So, for example, every precedes both everybody and everything, but comes after eve. Finally, if both sequences terminate at the same time and, in all cases, pair-wise characters match, then the two words are considered to be equal.

The lexicographical_compare() algorithm implements this idea, returning true if the first sequence is smaller than the second, and false otherwise. The algorithm has been generalized to any sequence. Thus the lexicographical_compare() algorithm can be used with arrays, strings, vectors, lists, or any of the other data structures used in the standard library.

```
bool lexicographical_compare
   (InputIterator first1, InputIterator last1,
   InputIterator first2, InputIterator last2 [, BinaryFunction ] );
```

Unlike most of the other algorithms that take two sequences as argument, the lexicographical_compare() algorithm uses a first and a past-end iterator for *both* sequences. A variation on the algorithm also takes a fifth argument,

which is the binary function used to compare corresponding elements from the two sequences.

The example program illustrates the use of this algorithm with character sequences, and with arrays of integer values.

```
void lexicographical_compare_example()
   // illustrate the use of the lexicographical_compare algorithm
{
   char * wordOne = "everything";
   char * wordTwo = "everybody";

   cout << "compare everybody to everything " <<
      lexicographical_compare(wordTwo, wordTwo + strlen(wordTwo),
         wordOne, wordOne + strlen(wordOne)) << endl;

   int a[] = {3, 4, 5, 2};
   int b[] = {3, 4, 5};
   int c[] = {3, 5};

   cout << "compare a to b:" <<
      lexicographical_compare(a, a+4, b, b+3) << endl;
   cout << "compare a to c:" <<
      lexicographical_compare(a, a+4, c, c+2) << endl;
}
```

13.7 Sequence-Generating Algorithms

Obtaining the Source

The example functions described in this section can be found in the file `alg6.cpp`.

The algorithms described in this section are all used to generate a new sequence from an existing sequence by performing some type of transformation. In most cases, the output sequence is described by an output iterator. This means these algorithms can be used to overwrite an existing structure (such as a ***vector***). Alternatively, by using an insert iterator (see Section 2.4), the algorithms can insert the new elements into a variable length structure, such as a ***set*** or ***list***. Finally, in some cases which we will note, the output iterator can be the same as one of the sequences specified by an input iterator, thereby providing the ability to make an in-place transformation.

The functions `partial_sum()` and `adjacent_difference()` are declared in the header file `numeric`, while the other functions are described in the header file `algorithm`.

13.7.1 Transform One or Two Sequences

The algorithm `transform()` is used either to make a general transformation of a single sequence, or to produce a new sequence by applying a binary function in a pair-wise fashion to corresponding elements from two different sequences. The general definition of the argument and result types are as follows:

```
OutputIterator transform (InputIterator first, InputIterator last,
   OutputIterator result, UnaryFunction);
OutputIterator transform
   (InputIterator first1, InputIterator last1,
```

```
   InputIterator first2,  OutputIterator result, BinaryFunction);
```

The first form applies a unary function to each element of a sequence. In the example program given below, this is used to produce a vector of integer values that hold the arithmetic negation of the values in a linked list. The input and output iterators can be the same, in which case the transformation is applied in-place, as shown in the example program.

The second form takes two sequences and applies the binary function in a pair-wise fashion to corresponding elements. The transaction assumes, but does not verify, that the second sequence has at least as many elements as the first sequence. Once more, the result can either be a third sequence, or either of the two input sequences.

```
int square(int n) { return n * n; }

void transform_example ()
// illustrate the use of the transform algorithm
{
// generate a list of value 1 to 6
   list<int> aList;
   generate_n (inserter(aList, aList.begin()), 6, iotaGen(1));

// transform elements by squaring, copy into vector
   vector<int> aVec(6);
   transform (aList.begin(), aList.end(), aVec.begin(), square);

// transform vector again, in place, yielding 4th powers
   transform (aVec.begin(), aVec.end(), aVec.begin(), square);

// transform in parallel, yielding cubes
   vector<int> cubes(6);
   transform (aVec.begin(), aVec.end(), aList.begin(),
      cubes.begin(), divides<int>());
}
```

13.7.2 Partial Sums

A partial sum of a sequence is a new sequence in which every element is formed by adding the values of all prior elements. For example, the partial sum of the vector 1 3 2 4 5 is the new vector 1 4 6 10 15. The element 4 is formed from the sum 1 + 3, the element 6 from the sum 1 + 3 + 2, and so on. Although the term “sum” is used in describing the operation, the binary function can, in fact, be any arbitrary function. The example program illustrates this by computing partial products. The arguments to the partial sum function are described as follows:

```
OutputIterator partial_sum
      (InputIterator first, InputIterator last,
       OutputIterator result [, BinaryFunction] );
```

By using the same value for both the input iterator and the result the partial sum can be changed into an in-place transformation.

```
void partial_sum_example ()
// illustrate the use of the partial sum algorithm
```

```
{
// generate values 1 to 5
   vector<int> aVec(5);
   generate (aVec.begin(), aVec.end(), iotaGen(1));

// output partial sums
   partial_sum (aVec.begin(), aVec.end(),
      ostream_iterator<int> (cout, " ")), cout << endl;

// output partial products
   partial_sum (aVec.begin(), aVec.end(),
      ostream_iterator<int> (cout, " "),
      times<int>() );
}
```

13.7.3 Adjacent Differences

An adjacent difference of a sequence is a new sequence formed by replacing every element with the difference between the element and the immediately preceding element. The first value in the new sequence remains unchanged. For example, a sequence such as (1, 3, 2, 4, 5) is transformed into (1, 3-1, 2-3, 4-2, 5-4), and in this manner becomes the sequence (1, 2, -1, 2, 1).

As with the algorithm `partial_sum()`, the term "difference" is not necessarily accurate, as an arbitrary binary function can be employed. The adjacent sums for this sequence are (1, 4, 5, 6, 9), for example. The adjacent difference algorithm has the following declaration:

```
OutputIterator adjacent_difference (InputIterator first,
   InputIterator last, OutputIterator result [, BinaryFunction ]);
```

By using the same iterator as both input and output iterator, the adjacent difference operation can be performed in place.

```
void adjacent_difference_example ()
// illustrate the use of the adjacent difference algorithm
{
// generate values 1 to 5
   vector<int> aVec(5);
   generate (aVec.begin(), aVec.end(), iotaGen(1));

// output adjacent differences
   adjacent_difference (aVec.begin(), aVec.end(),
      ostream_iterator<int,char> (cout, " ")), cout << endl;

// output adjacent sums
   adjacent_difference (aVec.begin(), aVec.end(),
      ostream_iterator<int,char> (cout, " "),
      plus<int>() );
}
```

13.8 Miscellaneous Algorithms

In the final section we describe the remaining algorithms found in the standard library.

13.8.1 Apply a Function to All Elements in a Collection

The algorithm `for_each()` takes three arguments. The first two provide the iterators that describe the sequence to be evaluated. The third is a one-argument function. The `for_each()` algorithm applies the function to each value of the sequence, passing the value as an argument.

```
Function for_each
   (InputIterator first, InputIterator last, Function);
```

For example, the following code fragment, which uses the `print_if_leap()` function, will print a list of the leap years that occur between 1900 and 1997:

```
cout << "leap years between 1990 and 1997 are: ";
for_each (1990, 1997, print_if_leap);
cout << endl;
```

The argument function is guaranteed to be invoked only once for each element in the sequence. The `for_each()` algorithm itself returns the value of the third argument, although this, too, is usually ignored.

Results Produced by Side Effect

The function passed as the third argument is not permitted to make any modifications to the sequence, so it can only achieve any result by means of a side effect, such as printing, assigning a value to a global or static variable, or invoking another function that produces a side effect. If the argument function returns any result, it is ignored.

The following example searches an array of integer values representing dates, to determine which vintage wine years were also leap years:

```
int vintageYears[] = {1947, 1955, 1960, 1967, 1994};
...

cout << "vintage years which were also leap years are: ";
for_each (vintageYears, vintageYears + 5, print_if_leap);
cout << endl;
```

Side effects need not be restricted to printing. Assume we have a function `countCaps()` that counts the occurrence of capital letters:

```
int capCount = 0;

void countCaps(char c)  { if (isupper(c)) capCount++; }
```

The following example counts the number of capital letters in a string value:

```
string advice = "Never Trust Anybody Over 30!";
for_each(advice.begin(), advice.end(),countCaps);
cout << "upper-case letter count is " << capCount << endl;
```

Section 14.

Ordered Collection Algorithms

14.1 Overview

In this section we will describe the generic algorithms in the standard library that are specific to ordered collections. These are summarized by the following table:

Name	Purpose
Sorting Algorithms Ð Sections 14.2 and 14.3	
sort	rearrange sequence, place in order
stable_sort	sort, retaining original order of equal elements
partial_sort	sort only part of sequence
partial_sort_copy	partial sort into copy
Find Nth largest Element Ð Section 14.4	
nth_element	locate nth largest element
Binary Search Ð Section 14.5	
binary_search	search, returning boolean
lower_bound	search, returning first position
upper_bound	search, returning last position
equal_range	search, returning both positions
Merge Ordered Sequences Ð Section 14.6	
merge	combine two ordered sequences
Set Operations Ð Section 14.7	
set_union	form union of two sets
set_intersection	form intersection of two sets
set_difference	form difference of two sets
set_symmetric_difference	form symmetric difference of two sets
includes	see if one set is a subset of another
Heap operations Ð Section 14.8	
make_heap	turn a sequence into a heap
push_heap	add a new value to the heap

Name	Purpose
pop_heap	remove largest value from the heap
sort_heap	turn heap into sorted collection

Ordered collections can be created using the standard library in a variety of ways. For example:

- The containers ***set***, ***multiset***, ***map*** and ***multimap*** are ordered collections by definition.
- A ***list*** can be ordered by invoking the `sort()` member function.
- A ***vector***, ***deque*** or ordinary C++ array can be ordered by using one of the sorting algorithms described later in this section.

Like the generic algorithms described in the previous section, the algorithms described here are not specific to any particular container class. This means they can be used with a wide variety of types. Many of them do, however, require the use of random-access iterators. For this reason they are most easily used with vectors, deques, or ordinary arrays.

Almost all the algorithms described in this section have two versions. The first version uses the `less than` operator (operator `<`) for comparisons appropriate to the container element type. The second, and more general, version uses an explicit comparison function object, which we will write as `Compare`. This function object must be a binary predicate (see Section 3.2). Since this argument is optional, we will write it within square brackets in the description of the argument types.

Obtaining the Sample Programs

The example programs described in this section have been combined and are included in the file `alg7.cpp` in the tutorial distribution. As we did in Section 13, we will generally omit output statements from the descriptions of the programs provided here, although they are included in the executable versions.

A sequence is considered to be ordered if for every valid (that is, denotable) iterator `i` with a denotable successor `j`, it is the case that the comparison `Compare(*j, *i)` is false. Note that this does not necessarily imply that `Compare(*i, *j)` is true. It is assumed that the relation imposed by `Compare` is transitive, and induces a total ordering on the values.

In the descriptions that follow, two values `x` and `y` are said to be equivalent if both `Compare(x, y)` and `Compare(y, x)` are false. Note that this need not imply that `x == y`.

14.1.1 Include Files

As with the algorithms described in Section 13, before you can use any of the algorithms described in this section in a program you must include the `algorithm` header file:

```
# include <algorithm>
```

14.2 Sorting Algorithms

There are two fundamental sorting algorithms provided by the standard library, described as follows:

```
void sort (RandomAccessIterator first,
      RandomAccessIterator last [, Compare ] );

void stable_sort (RandomAccessIterator first,
      RandomAccessIterator last [, Compare ] );
```

The `sort()` algorithm is slightly faster, but it does not guarantee that equal elements in the original sequence will retain their relative orderings in the final result. If order is important, then use the `stable_sort()` version.

Because these algorithms require random access iterators, they can be used only with vectors, deques, and ordinary C pointers. Note, however, that the list container provides its own `sort()` member function.

The comparison operator can be explicitly provided when the default operator `<` is not appropriate. This is used in the example program to sort a list into descending, rather than ascending order. An alternative technique for sorting an entire collection in the inverse direction is to describe the sequence using reverse iterators.

The following example program illustrates the `sort()` algorithm being applied to a ***vector***, and the `sort()` algorithm with an explicit comparison operator being used with a ***deque***.

More Sorts

Yet another sorting algorithm is provided by the heap operations, to be described in Section 14.8.

```
void sort_example ()
   // illustrate the use of the sort algorithm
{
      // fill both a vector and a deque
      // with random integers
   vector<int> aVec(15);
   deque<int> aDec(15);
   generate (aVec.begin(), aVec.end(), randomValue);
   generate (aDec.begin(), aDec.end(), randomValue);

      // sort the vector ascending
   sort (aVec.begin(), aVec.end());

      // sort the deque descending
   sort (aDec.begin(), aDec.end(), greater<int>() );

      // alternative way to sort descending
   sort (aVec.rbegin(), aVec.rend());
}
```

14.3 Partial Sort

The generic algorithm `partial_sort()` sorts only a portion of a sequence. In the first version of the algorithm, three iterators are used to describe the beginning, middle, and end of a sequence. If `n` represents the number of elements between the start and middle, then the smallest `n` elements will be

moved into this range in order. The remaining elements are moved into the second region. The order of the elements in this second region is undefined.

```
void partial_sort (RandomAccessIterator first,
   RandomAccessIterator middle,
   RandomAccessIterator last [ , Compare ]);
```

A second version of the algorithm leaves the input unchanged. The output area is described by a pair of random access iterators. If `n` represents the size of this area, then the smallest `n` elements in the input are moved into the output in order. If `n` is larger than the input, then the entire input is sorted and placed in the first `n` locations in the output. In either case the end of the output sequence is returned as the result of the operation.

```
RandomAccessIterator partial_sort_copy
   (InputIterator first, InputIterator last,
   RandomAccessIterator result_first,
   RandomAccessIterator result_last [, Compare ] );
```

Because the input to this version of the algorithm is specified only as a pair of input iterators, the `partial_sort_copy()` algorithm can be used with any of the containers in the standard library. In the example program it is used with a ***list***.

```
void partial_sort_example ()
      // illustrate the use of the partial sort algorithm
{
      // make a vector of 15 random integers
   vector<int> aVec(15);
   generate (aVec.begin(), aVec.end(), randomValue);

      // partial sort the first seven positions
   partial_sort (aVec.begin(), aVec.begin() + 7, aVec.end());

      // make a list of random integers
   list<int> aList(15, 0);
   generate (aList.begin(), aList.end(), randomValue);

      // sort only the first seven elements
   vector<int> start(7);
   partial_sort_copy (aList.begin(), aList.end(),
      start.begin(), start.end(), greater<int>());
}
```

14.4 nth Element

Imagine we have the sequence 2 5 3 4 7, and we want to discover the median, or middle element. We could do this with the function `nth_element()`. One result might be the following sequence:

3 2 | 4 | 7 5

The vertical bars are used to describe the separation of the result into three parts; the elements before the requested value, the requested value, and the values after the requested value. Note that the values in the first and third sequences are unordered; in fact, they can appear in the result in any order.

The only requirement is that the values in the first part are no larger than the value we are seeking, and the elements in the third part are no smaller than this value.

The three iterators provided as arguments to the algorithm `nth_element()` divide the argument sequence into the three sections we just described. These are the section prior to the middle iterator, the single value denoted by the middle iterator, and the region between the middle iterator and the end. Either the first or third of these may be empty.

The arguments to the algorithm can be described as follows:

```
void nth_element (RandomAccessIterator first,
   RandomAccessIterator nth,
   RandomAccessIterator last [, Compare ] );
```

Following the call on `nth_element(),` the nth largest value will be copied into the position denoted by the middle iterator. The region between the first iterator and the middle iterator will have values no larger than the nth element, while the region between the middle iterator and the end will hold values no smaller than the nth element.

The example program illustrates finding the fifth largest value in a vector of random numbers.

```
void nth_element_example ()
      // illustrate the use of the nth_element algorithm
{
      // make a vector of random integers
   vector<int> aVec(10);
   generate (aVec.begin(), aVec.end(), randomValue);

      // now find the 5th largest
   vector<int>::iterator nth = aVec.begin() + 4;
   nth_element (aVec.begin(), nth, aVec.end());

   cout << "fifth largest is " << *nth << endl;
}
```

14.5 Binary Search

The standard library provides a number of different variations on binary search algorithms. All will perform only approximately `log n` comparisons, where `n` is the number of elements in the range described by the arguments. The algorithms work best with random access iterators, such as those generated by vectors or deques, when they will also perform approximately `log n` operations in total. However, they will also work with non-random access iterators, such as those generated by lists, in which case they will perform a linear number of steps. Although legal, it is not necessary to perform a binary search on a ***set*** or ***multiset*** data structure, since those container classes provide their own search methods, which are more efficient.

The generic algorithm `binary_search()` returns `true` if the sequence contains a value that is equivalent to the argument. Recall that to be equivalent means that both `Compare(value, arg)` and `Compare(arg, value)` are false. The algorithm is declared as follows:

```
bool binary_search (ForwardIterator first, ForwardIterator last,
      const T & value [, Compare ] );
```

In other situations it is important to know the position of the matching value. This information is returned by a collection of algorithms, defined as follows:

```
   ForwardIterator lower_bound (ForwardIterator first,
      ForwardIterator last, const T& value [ , Compare ] );

   ForwardIterator upper_bound (ForwardIterator first,
      ForwardIterator last, const T& value [, Compare ] );

   pair<ForwardIterator, ForwardIterator> equal_range
      (ForwardIterator first, ForwardIterator last,
         const T& value [, Compare ] );
```

The algorithm `lower_bound()` returns, as an iterator, the first position into which the argument could be inserted without violating the ordering, whereas the algorithm `upper_bound()` finds the last such position. These will match only when the element is not currently found in the sequence. Both can be executed together in the algorithm `equal_range()`, which returns a pair of iterators.

Our example program shows these functions being used with a vector of random integers.

```
void binary_search_example ()
   // illustrate the use of the binary search algorithm
{
      // make an ordered vector of 15 random integers
   vector<int> aVec(15);
   generate (aVec.begin(), aVec.end(), randomValue);
   sort (aVec.begin(), aVec.end());

      // see if it contains an eleven
   if (binary_search (aVec.begin(), aVec.end(), 11))
      cout << "contains an 11" << endl;
   else
      cout << "does not contain an 11" << endl;

      // insert an 11 and a 14
   vector<int>::iterator where;
   where = lower_bound (aVec.begin(), aVec.end(), 11);
   aVec.insert (where, 11);

   where = upper_bound (aVec.begin(), aVec.end(), 14);
   aVec.insert (where, 14);

}
```

14.6 Merge Ordered Sequences

The algorithm `merge()` combines two ordered sequences to form a new ordered sequence. The size of the result is the sum of the sizes of the two argument sequences. This should be contrasted with the `set_union()` operation, which eliminates elements that are duplicated in both sets. The `set_union()` function will be described later in this section.

The merge operation is stable. This means, for equal elements in the two ranges, not only is the relative ordering of values from each range preserved, but the values from the first range always precede the elements from the second. The two ranges are described by a pair of iterators, whereas the result is defined by a single output iterator. The arguments are shown in the following declaration:

```
OutputIterator merge (InputIterator first1, InputIterator last1,
   InputIterator first2, InputIterator last2,
   OutputIterator result [, Compare ]);
```

The example program illustrates a simple merge, the use of a merge with an inserter, and the use of a merge with an output stream iterator.

```
void merge_example ()
   // illustrate the use of the merge algorithm
{
      // make a list and vector of 10 random integers
   vector<int> aVec(10);
   list<int> aList(10, 0);
   generate (aVec.begin(), aVec.end(), randomValue);
   sort (aVec.begin(), aVec.end());
   generate_n (aList.begin(), 10, randomValue);
   aList.sort();

      // merge into a vector
   vector<int> vResult (aVec.size() + aList.size());
   merge (aVec.begin(), aVec.end(), aList.begin(), aList.end(),
      vResult.begin());

      // merge into a list
   list<int> lResult;
   merge (aVec.begin(), aVec.end(), aList.begin(), aList.end(),
      inserter(lResult, lResult.begin()));

      // merge into the output
   merge (aVec.begin(), aVec.end(), aList.begin(), aList.end(),
      ostream_iterator<int,char> (cout, " "));
   cout << endl;

}
```

The algorithm `inplace_merge()` (Section 13.4.6) can be used to merge two sections of a single sequence into one sequence.

14.7 Set Operations

The operations of set union, set intersection, and set difference were all described in Section 8.2.7 when we discussed the ***set*** container class.

However, the algorithms that implement these operations are generic, and applicable to any ordered data structure. The algorithms assume the input ranges are ordered collections that represent ***multiset***s; that is, elements can be repeated. However, if the inputs represent ***set***s, then the result will always be a ***set***. That is, unlike the `merge()` algorithm, none of the set algorithms will produce repeated elements in the output that were not present in the input sets.

The set operations all have the same format. The two input sets are specified by pairs of input iterators. The output set is specified by an input iterator, and the end of this range is returned as the result value. An optional comparison operator is the final argument. In all cases it is required that the output sequence not overlap in any manner with either of the input sequences.

```
OutputIterator set_union
   (InputIterator first1, InputIterator last1,
   InputIterator first2, InputIterator last2,
   OutputIterator result [, Compare ] );
```

The example program illustrates the use of the four set algorithms, ***set_union***, ***set_intersection***, ***set_difference*** and ***set_symmetric_difference***. It also shows a call on `merge()` in order to contrast the merge and the set union operations. The algorithm `includes()` is slightly different. Again the two input sets are specified by pairs of input iterators, and the comparison operator is an optional fifth argument. The return value for the algorithm is `true` if the first set is entirely included in the second, and `false` otherwise.

```
void set_example ()
   // illustrate the use of the generic set algorithms
{
   ostream_iterator<int> intOut (cout, " ");

      // make a couple of ordered lists
   list<int> listOne, listTwo;
   generate_n (inserter(listOne, listOne.begin()), 5, iotaGen(1));
   generate_n (inserter(listTwo, listTwo.begin()), 5, iotaGen(3));

      // now do the set operations
      // union - 1 2 3 4 5 6 7
   set_union (listOne.begin(), listOne.end(),
      listTwo.begin(), listTwo.end(), intOut), cout << endl;
      // merge - 1 2 3 3 4 4 5 5 6 7
   merge (listOne.begin(), listOne.end(),
      listTwo.begin(), listTwo.end(), intOut), cout << endl;
      // intersection - 3 4 5
   set_intersection (listOne.begin(), listOne.end(),
      listTwo.begin(), listTwo.end(), intOut), cout << endl;
      // difference - 1 2
   set_difference (listOne.begin(), listOne.end(),
      listTwo.begin(), listTwo.end(), intOut), cout << endl;
      // symmetric difference - 1 2 6 7
   set_symmetric_difference (listOne.begin(), listOne.end(),
      listTwo.begin(), listTwo.end(), intOut), cout << endl;

   if (includes (listOne.begin(), listOne.end(),
```

```
      listTwo.begin(), listTwo.end()))
         cout << "set is subset" << endl;
   else
      cout << "set is not subset" << endl;
}
```

14.8 Heap Operations

A *heap* is a binary tree in which every node is larger than the values associated with either child. A heap (and, for that matter, a binary tree) can be very efficiently stored in a vector, by placing the children of node `i` in positions `2 * i + 1` and `2 * i + 2`.

Using this encoding, the largest value in the heap will always be located in the initial position, and can therefore be very efficiently retrieved. In addition, efficient (logarithmic) algorithms exist that both permit a new element to be added to a heap and the largest element removed from a heap. For these reasons, a heap is a natural representation for the ***priority_queue*** data type, described in Section 11.

The default operator is the less-than operator (operator `<`) appropriate to the element type. If desired, an alternative operator can be specified. For example, by using the greater-than operator (operator `>`), one can construct a heap that will locate the smallest element in the first location, instead of the largest.

The algorithm `make_heap()` takes a range, specified by random access iterators, and converts it into a heap. The number of steps required is a linear function of the number of elements in the range.

```
void make_heap (RandomAccessIterator first,
      RandomAccessIterator last [, Compare ]);
```

A new element is added to a heap by inserting it at the end of a range (using the `push_back()` member function of a vector or deque, for example), followed by an invocation of the algorithm `push_heap()`. The `push_heap()` algorithm restores the heap property, performing at most a logarithmic number of operations.

```
void push_heap (RandomAccessIterator first,
      RandomAccessIterator last [, Compare ]);
```

The algorithm `pop_heap()` swaps the first and final elements in a range, then restores to a heap the collection without the final element. The largest value of the original collection is therefore still available as the last element in the range (accessible, for example, using the `back()` member function in a vector, and removable using the `pop_back()` member function), while the remainder of the collection continues to have the heap property. The `pop_heap()` algorithm performs at most a logarithmic number of operations.

```
void pop_heap (RandomAccessIterator first,
      RandomAccessIterator last [, Compare ]);
```

Heaps and Ordered Collections

Note that an ordered collection is a heap, but a heap need not necessarily be an ordered collection. In fact, a heap can be constructed in a sequence much more quickly than the sequence can be sorted.

Finally, the algorithm sort_heap() converts a heap into a ordered (sorted) collection. Note that a sorted collection is still a heap, although the reverse is not the case. The sort is performed using approximately n log n operations, where n represents the number of elements in the range. The sort_heap() algorithm is not stable.

```
void sort_heap (RandomAccessIterator first,
      RandomAccessIterator last [, Compare ]);
```

Here is an example program that illustrates the use of these functions.

```
void heap_example ()
   // illustrate the use of the heap algorithms
{
      // make a heap of 15 random integers
   vector<int> aVec(15);
   generate (aVec.begin(), aVec.end(), randomValue);
   make_heap (aVec.begin(), aVec.end());
   cout << "Largest value " << aVec.front() << endl;

      // remove largest and reheap
   pop_heap (aVec.begin(), aVec.end());
   aVec.pop_back();

      // add a 97 to the heap
   aVec.push_back (97);
   push_heap (aVec.begin(), aVec.end());

      // finally, make into a sorted collection
   sort_heap (aVec.begin(), aVec.end());
}
```

Section 15. Using Allocators

15.1 An Overview of the Standard Library Allocators

The Standard C++ allocator interface encapsulates the types and functions needed to manage the storage of data in a generic way. The interface provides:

- pointer and reference types;
- the type of the difference between pointers;
- the type for any object's `size`;
- storage allocation and deallocation primitives;
- object construction and destruction primitives.

The allocator interface wraps the mechanism for managing data storage, and separates this mechanism from the classes and functions used to maintain associations between data elements. This eliminates the need to rewrite containers and algorithms to suit different storage mechanisms. The interface lets you encapsulate all the storage mechanism details in an allocator, then provide that allocator to an existing container when appropriate.

The Standard C++ Library provides a default allocator class, ***allocator***, that implements this interface using the Standard `new` and `delete` operators for all storage management.

This section briefly describes how to use allocators with existing containers, then discusses what you need to consider when designing your own allocators. The later section of this guide, entitled "Building Containers and Generic Algorithms" describes what you must consider when designing containers that use allocators.

15.2 Using Allocators with Existing Standard Library Containers

Using allocators with existing Standard C++ Library container classes is a simple process. Merely provide an allocator type when you instantiate a container, and provide an actual allocator object when you construct a container object:

```
my_allocator alloc<int>;  // Construct an allocator
// Use the allocator
vector<int,my_allocator<int> > v(alloc);
```

All standard containers default the allocator template parameter type to `allocator<T>` and the object to `Allocator()`, where `Allocator` is the template parameter type. This means that the simplest use of allocators is to ignore them entirely. When you do not specify an allocator, the default allocator will be used for all storage management.

If you do provide a different allocator type as a template parameter, then the type of object that you provide must match the template type. For example, the following code will cause an compiler error because the types in the template signature and the call to the allocator constructor don't match:

```
template <class T> class my_alloc;
list <int, allocator<int> > my_list(my_alloc()); \\ Wrong!
```

The following call to the allocator constructor does match the template signature:

```
list <int, my_alloc<int> > my_list(my_alloc());
```

Note that the container always holds a *copy* of the allocator object that is passed to the constructor. If you need a single allocator object to manage all storage for a number of containers, you must provide an allocator that maintains a reference to some shared implementation.

15.3 Building Your Own Allocators

Defining your own allocator is a relatively simple process. The Standard C++ Library describes a particular interface, consisting of types and functions. An allocator that conforms to the Standard must match the syntactic requirements for these member functions and types. The Standard C++ Library also specifies a portion of the semantics for the allocator type.

The Standard C++ Library allocator interface relies heavily on member templates. As of this writing, many compilers do not yet support both member function templates and member class templates. This makes it impossible to implement a standard allocator. Rogue Wave's implementation of the Standard C++ Library provides an alternative allocator interface that provides most of the power of the standard interface, without requiring unavailable compiler features. This interface differs significantly from the standard interface, and will not work with other vendors' versions of the Standard C++ Library.

We recommend that when you define an allocator and implement containers, you provide both the standard interface and the Rogue Wave interface. This will allow you to use allocators now, and to take advantage of the standard once it becomes available on your compiler.

The remainder of this section describes the requirements for the Standard C++ Library allocator, the requirements for Rogue Wave's alternative allocator, and some techniques that specify how to support both interfaces in the same code base.

15.3.1 Using the Standard Allocator Interface

An allocator that conforms to the Standard C++ Library allocator specification must have the following interface. The example uses `my_allocator` as a place holder for your own allocator name:

```
template <class T>
class my_allocator
{
  typedef implementation_defined size_type;
  typedef implementation_defined difference_type
  typedef implementation_defined pointer;
  typedef implementation_defined const_pointer;
  typedef implementation_defined reference;
  typedef implementation_defined const_reference;
  typedef implementation_defined value_type;

  template <class U>
  struct rebind { typedef allocator<U> other; };
```

Each of the pointer types in this interface must have a conversion to `void*`. It must be possible to use the resulting `void*` as a `this` value in a constructor or destructor and in conversions to `B<void>::pointer` (for appropriate `B`) for use by `B::deallocate()`.

The rebind member allows a container to construct an allocator for some arbitrary type out of the allocator type provided as a template parameter. For instance, the list container gets an allocator<T> by default, but a list may well need to allocate list_nodes as well as T's. The container can construct an allocator for list_nodes out of the allocator for T's (the template parameter, Allocator, in this case) as follows:

```
Allocator::rebind<list_node>::other list_node_allocator;
```

Here is a description of the member functions that a Standard C++ Library allocator must provide:

```
my_allocator();
template <class U>
my_allocator(const my_allocator<U>&);
template <class U>
operator=(const my_allocator<U>&);

~my_allocator();
```

Constructors and destructor.

```
pointer address(reference r) const;
```

Returns the address of `r` as a `pointer` type. This function and the following function are used to convert references to pointers.

```
const_pointer address(const_reference r)
                        const;
```

Returns the address of `r` as a `const_pointer` type.

```
pointer allocate(size_type n);
```

Allocate storage for `n` values of `T`.

```
template <class T, class U>
types<T>::pointer allocate(size_type n,
    typename my_allocator<void>::const_pointer hint = 0);
```

Allocate storage for `hint` values of `T`, using the value of `u` as an implementation-defined hint for determining the best storage placement.

```
void
deallocate(pointer);
```
Deallocate storage obtained by a call to `allocate`.

```
size_type
max_size();
```
Return the largest possible storage available through a call to `allocate`.

```
void
construct(pointer p, const T& val);
```
Construct an object of type `T` at the location of `p`, using the value of `u` in the call to the constructor for `T`. The effect is:

```
new((void*)p) T(u);
```

```
void
destroy(pointer p);
```
Call the destructor on the value pointed to by `p`. The effect is:

```
((T*)p)->~T()
```

Here is a description of the non-member functions that a Standard C++ Library allocator must provide:

```
template <class T>
my_allocator::pointer
operator new(my_allocator::size_type, my_allocator&);
```
Allocate space for a single object of type T using `my_allocator::allocate`. The effect is:

```
new((void*)x.template allocate<T>(1)) T;
```

```
template <class T, class U>
bool
operator==(const my_allocator<T>& a,
           const my_allocator<U>& b);
```
Return `true` if allocators `b` and `a` can be safely interchanged. "Safely interchanged" means that `b` could be used to deallocate storage obtained through `a` and vice versa.

```
template <class T, class U>
bool
operator!=(const my_allocator<T>& a,
           const my_allocator<U>& b);
```
Return `!(a == b)`.

15.3.2 Using Rogue Wave's Alternative Interface

Rogue Wave provides an alternative allocator interface for those compilers that do not support both class templates and member function templates.

In this interface, the class ***allocator_interface*** provides all types and typed functions. Memory is allocated as raw bytes using the class provide by the

`Allocator` template parameter. Functions within ***allocator_interface*** cast appropriately before returning pointer values. Because multiple ***allocator_interface*** objects can attach to a single allocator, one allocator can allocate all storage for a container, regardless of how many types are involved. The one real restriction is that pointers and references are hard-coded as type `T*` and `T&`. (Note that in the standard interface they are `implementation_defined`.). If your compiler supports partial specialization instead of member templates you can use it to get around even this restriction by specializing ***allocator_interface*** on just the allocator type.

To implement an allocator based on the alternative interface, supply the class labeled `my_allocator` below.

```
//
// Alternative allocator uses an interface class
// (allocator_interface)
// to get type safety.
//
template <class T>
class  my_allocator
{
  public:
    typedef implementation_defined size_type;
    typedef implementation_defined difference_type;
    typedef implementation_defined pointer;
    typedef implementation_defined const_pointer;
    typedef implementation_defined reference;
    typedef implementation_defined const_reference;
    typedef implementation_defined value_type;

    my_allocator();
    ~my_allocator();

    void * allocate (size_type n, void *  = 0);
    void deallocate (void* p);
    size_type max_size (size_type size) const
};
```

We've also included a listing of the full implementation of the ***allocator_interface*** class, to show how a standard container will use your class. The section entitled "Building Containers & Generic Algorithms" provides a full description of how the containers use the alternative interface.

```
template <class Allocator,class T>
class allocator_interface
{
public:
  typedef Allocator allocator_type;
  typedef T*         pointer;
  typedef const T*   const_pointer;
  typedef T&         reference;
  typedef const T&   const_reference;
  typedef T          value_type;
  typedef typename Allocator::size_type    size_type;
  typedef typename Allocator::difference_type difference_type;

protected:
  allocator_type*         alloc_;

public:
  allocator_interface() : alloc_(0) { ; }
```

```
  allocator_interface(Allocator* a) : alloc_(a) { ; }
  void alloc(Allocator* a)
  {
    alloc_ = a;
  }
  pointer address (T& x)
  {
    return static_cast<pointer>(&x);
  }
  size_type max_size ()  const
  {
    return alloc_->max_size(sizeof(T));
  }
  pointer allocate(size_type n, pointer  = 0)
  {
    return static_cast<pointer>(alloc_->allocate(n*sizeof(T)));
  }
  void deallocate(pointer p)
  {
    alloc_->deallocate(p);
  }
  void construct(pointer p, const T& val)
  {
    new (p) T(val);
  }
  void destroy(T* p)
  {
    ((T*)p)->~T();
  }
};
class allocator_interface<my_allocator,void>
{
  public:
    typedef void*         pointer;
    typedef const void*   const_pointer;
};
//
// allocator globals
//
void * operator new(size_t N, my_allocator& a);
inline void * operator new[](size_t N, my_allocator& a);
inline bool operator==(const my_allocator&, const my_allocator&);
```

15.3.3 How to Support Both Interfaces

Rogue Wave strongly recommends that you implement containers that support both the Standard C++ Library allocator interface, and our alternative interface. By supporting both interfaces, you can use allocators now, and take advantage of the standard once it becomes available on your compiler.

In order to implement both versions of the allocator interface, your containers must have some mechanism for determining whether the standard interface is available. Rogue Wave provides the macro `_RWSTD_ALLOCATOR` in `stdcomp.h` to define whether or not the standard

allocator is available. If _RWSTD_ALLOCATOR evaluates to true, your compiler is capable of handling Standard C++ Library allocators, otherwise you must use the alternative.

The first place that you use _RWSTD_ALLOCATOR is when determining which typenames the container must use to reflect the interface. To do this, place the equivalent of the following code in your container class definition:

```
#ifdef RWSTD_ALLOCATOR
    typedef typename Allocator::rebind<T>::other::reference
         reference;
    typedef typename
         Allocator::rebind<T>::other::const_reference
         const_reference;
    typedef typename Allocator::rebind<node>::other::pointer
         link_type;

    typedef Allocator::rebind<T>::other value_allocator;
    typedef Allocator::rebind<node>::other node_allocator;
#else
    typedef typename
       allocator_interface<Allocator,T>::reference reference;
    typedef typename
       allocator_interface<Allocator,T>::const_reference
         const_reference;
    typedef typename
       allocator_interface<Allocator,node>::pointer  link_type;

  Allocator alloc;
  typedef allocator_interface<Allocator,T>  value_allocator;
  typedef allocator_interface<Allocator,node>
                                            node_allocator;
#endif
```

Notice that we use rebind even for the types associated with T. This is safest since it ensures that the container will work even if the allocator is instantiated with a different type for the allocator template parameter(say vector<int, allocator<void> >. This makes our containers more robust. Note also that we provide two allocator types: value_allocator and node_allocator. You will need to assemble actual allocators inside your container, probably as they're needed. In our example, the mechanism for calling allocator::allocate for T's looks like this (regardless which interface is being used):

```
value_allocator(alloc)::allocate(…);
```

In this call we are constructing an appropriate allocator using its template copy constructor and then we call allocate on that allocator. One result of this use of the allocator is that any state held by an allocator had better be passed through the copy constructor by reference, so that it is maintained in the one allocator object that we keep around (the one passed into the constructor for the container).

Section 16.
Building Containers & Generic Algorithms

16.1 Extending the Library

The adoption of the Standard Library for C++ marks a very important development for users of the C++ programming language. Although the library is written in an OOP language and provides plenty of objects, it also employs an entirely different paradigm. This other approach, called "generic programming," provides a flexible way to apply generic algorithms to a wide variety of different data structures. The flexibility of C++ in combination with this synthesis of two advanced design paradigms results in an unusual and highly-extensible library.

The clearest example of this synthesis is the ability to extend the library with user-defined containers and algorithms. This extension is possible because the definition of data structures has been separated from the definition of generic operations on those structures. The library defines very specific parameters for these two broad groups, giving users some confidence that containers and algorithms from different sources will work together as long as they all meet the specifications of the standard. At the same time, containers encapsulate data and a limited range of operations on that data in classic OOP fashion.

Each standard container is categorized as one of two types: a sequence or an associative container. A user-defined container need not fit into either of these two groups since the standard also defines rudimentary requirements for a container, but the categorization can be very useful for determining which algorithms will work with a particular container and how efficiently those algorithms will work. In determining the category of a container, the most important characteristics are the *iterator category* and *element ordering*. (The *Tutorial and Reference Guide* sections on each container describe the container types and iterator categories.)

Standard algorithms can be grouped into categories using a number of different criteria. The most important of these are: 1) whether or not the algorithm modifies the contents of a container; 2) the type of iterator required by the algorithm; and 3) whether or not the algorithm requires a container to be sorted. An algorithm may also require further state conditions from any container it's applied to. For instance, all the standard set algorithms not only require that a container be in sorted order, but also that the order of elements was determined using the same compare function or object that will be used by the algorithm.

16.2 Building on the Standard Containers

Let's examine a few of the ways you can use existing Standard C++ Library containers to create your own containers. For example, say you want to implement a set container that enforces unique values that are not inherently sorted. You also want a group of algorithms to operate on that set. The container is certainly a sequence, but not an associative container, since an

associative container is, by definition, sorted. The algorithms will presumably work on other sequences, assuming those sequences provide appropriate iterator types, since the iterator required by a set of algorithms determines the range of containers those algorithms can be applied to. The algorithms will be universally available if they only require forward iterators. On the other hand, they'll be most restrictive if they require random access iterators.

Simple implementations of this set container could make use existing Standard Library containers for much of their mechanics. Three possible ways of achieving this code re-use are:

- Inheritance;
- Generic inheritance;
- Generic composition.

Let's take a look at each of these approaches.

16.2.1 Inheritance

The new container could derive from an existing standard container, then override certain functions to get the desired behavior. One approach would be to derive from the ***vector*** container, as shown here:

```
#include <vector>

// note the use of a namespace to avoid conflicts with standard //
or global names

namespace my_namespace {

template <class T, class Allocator = std::allocator>
class set : public std::vector<T,Allocator>
{
public:
// override functions such as insert
  iterator insert (iterator position, const T& x)
  {
      if (find(begin(),end(),x) == end())
        return vector<T,Allocator>::insert(position,x);
      else
        return end(); // This value already present!
  }
…

};

} // End of my_namespace
```

16.2.2 Generic Inheritance

A second approach is to create a generic adaptor, rather than specifying ***vector***. You do this by providing the underlying container through a template parameter:

```
namespace my_namespace {

template <class T, class Container = std::vector<T> >
class set : public Container
{
public:
// Provide typedefs (iterator only for illustration)
  typedef  typename Container::iterator iterator;

// override functions such as insert
  iterator insert (iterator position, const T& x)
  {
      if (find(begin(),end(),x) == end())
        return Container::insert(position,x);
      else
        return end();  // This value already present!
  }
…

};

} // End of my_namespace
```

If you use generic inheritance through an adaptor, the adaptor and users of the adaptor cannot expect more than default capabilities and behavior from any container used to instantiate it. If the adaptor or its users expect functionality beyond what is required of a basic container, the documentation must specify precisely what is expected.

16.2.3 Generic Composition

The third approach uses *composition* rather than inheritance. (You can see the spirit of this approach in the Standard adaptors ***queue***, ***priority_queue*** and ***stack***.) When you use generic composition, you have to implement all of the desired interface. This option is most useful when you want to limit the behavior of an adaptor by providing only a subset of the interface provided by the container.

```
namespace my_namespace {

template <class T, class Container = std::vector<T> >
class set
{
protected:
   Container c;
public:
// Provide needed typedefs
  typedef  typename Container::iterator iterator;
```

```
// provide all necessary functions such as insert
  iterator insert (iterator position, const T& x)
  {
      if (find(c.begin(),c.end(),x) == c.end())
        return c.insert(position,x);
      else
        return c.end();  // This value already present!
  }
…

};

} // End of my_namespace
```

The advantages of adapting existing containers are numerous. For instance, you get to reuse the implementation and reuse the specifications of the container that you're adapting.

16.3 Creating Your Own Containers

All of the options that build on existing Standard C++ Library containers incur a certain amount of overhead. When performance demands are critical, or the container requirements very specific, there may be no choice but to implement a container from scratch.

When building from scratch, there are three sets of design requirements that you must meet:

- Container interface requirements;
- Allocator interface requirements;
- Iterator requirements.

We'll talk about each of these below.

16.3.1 Meeting the Container Requirements

The Standard C++ Library defines general interface requirements for containers, and specific requirements for specialized containers. When you create a container, the first part of your task is making sure that the basic interface requirements for a container are met. In addition, if your container will be a sequence or an associative container, you need to provide all additional pieces specified for those categories. For anything but the simplest container, this is definitely not a task for the faint of heart.

It’s very important to meet the requirements so that users of the container will know exactly what capabilities to expect without having to read the code directly. Review the sections on individual containers for information about the container requirements.

16.3.2 Meeting the Allocator Interface Requirements

A user-defined container will make use of the `allocator` interface for all storage management. (An exception to this is a container that will exist in a completely self-contained environment where there will be no need for substitute allocators.)

The basic interface of an allocator class consists of a set of typedefs, a pair of allocation functions, `allocate` and `deallocate`, and a pair of construction / destruction members, `construct` and `destroy`. The typedefs are used by a container to determine what pointers, references, sizes and differences look like. (A difference is a distance between two pointers.) The functions are used to do the actual management of data storage.

To use the allocator interface, a container must meet the following three requirements.

1. A container needs to have a set of typedefs that look like the following:

```
typedef Allocator allocator_type;
typedef typename Allocator::size_type size_type;
typedef typename Allocator::difference_type  difference_type;
typedef typename Allocator::reference reference;
typedef typename Allocator::const_reference
         const_reference;
typedef implementation_defined iterator;
typedef implementation_defined iterator;
```

2. A container also needs to have an `Allocator` member that will contain a copy the allocator argument provided by the constructors.

```
protected:
Allocator the_allocator;
```

3. A container needs to use that `Allocator` member for all storage management. For instance, our ***set*** container might have a naïve implementation that simply allocates a large buffer and then constructs values on that buffer. Note that this not a very efficient mechanism, but it serves as a simple example. We're also going to avoid the issue of `Allocator::allocate` throwing an exception, in the interest of brevity.

An abbreviated version of the ***set*** class appears below. The class interface shows the required typedefs and the `Allocator` member for this class.

```
#include <memory>

namespace my_namespace {

template <class T, class Allocator = std::allocator<T> >
class set
{
public:
   // typedefs and allocator member as  above
   typedef Allocator allocator_type;
   typedef typename Allocator::size_type size_type;
   typedef typename Allocator::difference_type
                              difference_type;
   typedef typename Allocator::reference reference;
```

```
    typedef typename Allocator::const_reference
                                          const_reference;

    // Our iterator will be a simple pointer
    typedef Allocator::pointer iterator;
    typedef Allocator::const_pointer iterator;

  protected:
    Allocator the_allocator;  // copy of the allocator

  private:
    size_type buffer_size;
    iterator buffer_start;
    iterator current_end;
    iterator end_of_buffer;

  public:
    // A constructor that initializes the set using a range
    // from some other container or array
    template <class Iterator>
    set(Iterator start, Iterator finish,
        Allocator alloc = Allocator());

    iterator begin() { return buffer_start; }
    iterator end() { return current_end; }
  };
```

Given this class interface, here's a definition of a possible constructor that uses the allocator. The numbered comments following this code briefly describe the allocator's role. For a fuller treatment of allocators take a look at the *Tutorial and Class Reference* sections for allocators.

```
  template <class T, class Allocator>
  template <class Iterator>
  set<T,Allocator>::set(Iterator start, Iterator finish,
      Allocator alloc)
    : buffer_size(finish-start + DEFAULT_CUSHION),
      buffer_start(0),
      current_end(0), end_of_buffer(0)
  {
    // copy the argument to our internal object
    the_allocator = alloc;                                  // 1

    // Create an initial buffer
    buffer_start = the_allocator.allocate(buffer_size);  // 2
    end_of_buffer = buffer_start + buffer_size;

    // construct new values from iterator range on the buffer
    for (current_end = buffer_start;
         start != finish;
         current_end++, start++)
       the_allocator.construct(current_end,*start);      // 3

    // Now let's remove duplicates using a standard algorithm
    std::unique(begin(),end());
  }


  } // End of my_namespace
```

`//1` The allocator parameter is copied into a protected member of the container. This private copy can then be used for all subsequent storage management.

`//2` An initial buffer is allocated using the allocator's allocate function.

`//3` The contents of the buffer are initialized using the values from the iterator range supplied to the constructor by the `start` and `finish` parameters. The `construct` function constructs an object at a particular location. In this case the location is at an index in the container's buffer.

16.3.3 Iterator Requirements

Every container must define an iterator type. Iterators allow algorithms to iterate over the container's contents. Although iterators can range from simple to very complex, it is the *iterator category*, not the complexity, that most affects an algorithm. The iterator category describes capabilities of the iterator, such as which direction it can traverse. The "Tips and Techniques" section below, and the iterator entries in the reference provides additional information about iterator categories.

The example in the previous section shows the implementation of a container that uses a simple pointer. A simple pointer is actually an example of the most powerful type of iterator: a random access iterator. If an iterator supports random access, we can add to or subtract from it as easily as we can increment it.

Some iterators have much less capability. For example , consider an iterator attached to a singly-linked list. Since each node in the list has links leading forward only, a naïve iterator can advance through the container in only one direction. An iterator with this limitation falls into the category of forward iterator.

Certain member functions such as `begin()`and `end()` produce iterators for a container. A container's description should always describe the category of iterator that its member functions produce. That way, a user of the container can see immediately which algorithms can operate successfully on the container.

16.4 Tips and Techniques for Building Algorithms

This sections describes some techniques that use features of iterators to increase the flexibility and efficiency of your algorithms.

16.4.1 The iterator_category Primitive

Sometimes an algorithm that can be implemented most efficiently with a random access iterator can also work with less powerful iterators. The Standard C++ Library includes primitives that allow a single algorithm to

provide several different implementations, depending upon the power of the iterator passed into it. The following example demonstrates the usual technique for setting up multiple versions of the same algorithm.

```
// Note, this requires that the iterators be derived from
// Standard base types, unless the iterators are simple pointers.

namespace my_namespace {

template <class Iterator>
Iterator union(Iterator first1, Iterator last1,
               Iterator first2, Iterator last2,
               Iterator Result)
{
  return union_aux(first1,last1,first2,last2,Result,
                   iterator_category(first1));
}

template <class Iterator>
Iterator union_aux(Iterator first1, Iterator last1,
               Iterator first2, Iterator last2,
               Iterator Result, forward_iterator_tag)
{
  // General but less efficient implementation
}

template <class Iterator>
Iterator union_aux(Iterator first1, Iterator last1,
               Iterator first2, Iterator last2,
               Iterator Result,
               random_access_iterator_tag)
{
  // More efficient implementation
}

} // End of my_namespace
```

The iterator primitive `iterator_category()` returns a tag that selects and uses the best available implementation of the algorithm. In order for `iterator_category()` to work, the iterator provided to the algorithm must be either a simple pointer type, or derived from one of the basic Standard C++ Library iterator types. When you use the `iterator_category()` primitive, the default implementation of the algorithm should expect at most a forward iterator. This default version will be used if the algorithm encounters an iterator that is not a simple pointer or derived from a basic standard iterator. (Note that input and output iterators are less capable than forward iterators, but that the requirements of an algorithms generally mandate read / write capabilities.)

16.4.2 The distance and advance Primitives

The `value_type` primitive lets you determine the type of value pointed to by an iterator. Similarly, you can use the `distance_type` primitive to get a type that represents distances between iterators.

In order to efficiently find the distance between two iterators, regardless of their capabilities, you can use the distance primitive. The distance primitive uses the technique shown above to send a calling program to one of four different implementations. This offers a considerable gain in efficiency, since an implementation for a forward iterator must step through the range defined by the two iterators:

```
Distance d = 0;
while (start++ != end)
  d++;
```

whereas an implementation for a random access iterator can simply subtract the start iterator from the end iterator:

```
Distance d = end - start;
```

Similar gains are available with the advance primitive, which allows you to step forward (or backward) an arbitrary number of steps as efficiently as possible for a particular iterator.

Section 17. The Traits Parameter

Consider the following problem. You have a matrix that must work for all types of numbers, but the behavior of the matrix depends, in at least some measure, on the type of `number`. This means your matrix can't handle all numbers in the same way.

Except for the behavioral difference, it sounds like the perfect problem for a template. But you can't hang extra information on the number type because it's often just a built-in type, so you can't use a single template. The template will do the same thing for every number type, which is just what we can't do in this case. You could specialize, but then you have to re-implement the entire matrix class for every type of number. It may well be that most of the class is the same. Worse yet, if you want to leave your interface open for use with some unknown future type, you're requiring that future user to reimplement the entire class as well.

What you really want is to put everything that doesn't change in one place, and repeatedly specify only the small part that does change with the type. The technique for doing this is generally referred to as using a *traits parameter*.

17.1 Using the Traits Technique

To implement a traits parameter for a class, you add it as an extra template parameter to your class. You then supply a class for this parameter that encapsulates all the specific operations. Usually that class is itself a template.

As an example, let's look at the matrix problem described above. By using the traits technique, when you want to add a new type to the matrix you simply specialize the traits class, not the entire matrix. You do no more work than you have to and retain the ability to use the matrix on any reasonable number.

Here's how the matrix traits template and specializations for `long` and `int` might look. The example also includes a skeleton of the matrix class that uses the traits template.

```
template <class Num>
class matrix_traits
{
  // traits functions and literals
};
template <class Num, class traits>
class matrix
{
  // matrix
}

class matrix_traits<long>
{
  // traits functions and literals specific to long
};

class matrix_traits<int>
{
```

```
  // traits functions and literals specific to int
};

… etc.

matrix<int, matrix_traits<int> > int_matrix;
matrix<long, matrix_traits<long> > long_matrix;
```

Of course you don't even have to specialize on `matrix_traits`. You just have to make sure you provide the interface that `matrix` expects from its traits template parameter.

Most of the time, the operations contained in a traits class will be static functions so that there's no need to actually instantiate a `traits` object.

The Standard Library uses this technique to give the ***string*** class maximum flexibility and efficiency across a wide range of types. The ***char_traits*** traits class provides elementary operations on character arrays. In the simplest case, this means providing ***string*** a `wstring` with access to the 'C' library functions for skinny and wide characters, for example `Strcpy` and `wcstrcpy`.

See the ***char_traits*** reference entry for a complete description of the traits class.

Section 18. Exception Handling

18.1 Overview

The Standard C++ Library provides a set of classes for reporting errors. These classes use the exception handling facility of the language. The library implements a particular error model, which divides errors in two broad categories: logic errors and runtime errors.

Logic errors are errors caused by problems in the internal logic of the program. They are generally preventable.

Runtime errors, on the other hand, are generally not preventable, or at least not predictable. These are errors generated by circumstances outside the control of the program, such as peripheral hardware faults.

18.1.1 Include Files

```
#include <stdexcept>
```

18.2 The Standard Exception Hierarchy

The library implements the two-category error model described above with a set of classes. These classes are defined in the `stdexcept` header file. They can be used to catch exceptions thrown by the library and to throw exceptions from your own code.

The classes are related through inheritance. The inheritance hierarchy looks like this:

exception

logic_error

domain_error

invalid_argument

length_error

out_of_range

runtime_error

range_error

overflow_error

underflow_error

Classes ***logic_error*** and ***runtime_error*** inherit from class ***exception***. All other exception classes inherit from either ***logic_error*** or ***runtime_error***.

18.3 Using Exceptions

All exceptions thrown explicitly by any element of the library are guaranteed to be part of the standard exception hierarchy. Review the reference for these classes to determine which functions throw which exceptions. You can then choose to catch particular exceptions, or catch any that might be thrown (by specifying the base class exception).

For instance, if you are going to call the `insert` function on ***string*** with a position value that could at some point be invalid, then you should use code like this:

```
string s;
int n;
...
try
{
s.insert(n,"Howdy");
}
catch (const exception& e)
{
   // deal with the exception
}
```

To throw your own exceptions, simply construct an exception of an appropriate type, assign it an appropriate message and throw it. For example:

```
...
if (n > max)
   throw out_of_range("Your past the end, bud");
```

The class ***exception*** serves as the base class for all other exception classes. As such it defines a standard interface. This interface includes the `what()` member function, which returns a null-terminated string that represents the message that was thrown with the exception. This function is likely to be most useful in a catch clause, as demonstrated in the example program at the end of this section.

The class ***exception*** does not contain a constructor that takes a message string, although it can be thrown without a message. Calling `what()` on an exception object will return a default message. All classes derived from ***exception*** *do* provide a constructor that allows you to specify a particular message.

To throw a base exception you would use the following code:

```
throw exception;
```

This is generally not very useful, since whatever catches this exception will have no idea what kind of error has occurred. Instead of a base exception, you will usually throw a derived class such as ***logic_error*** or one of its derivations (such as ***out_of_range*** as shown in the example above). Better still, you can extend the hierarchy by deriving your own classes. This allows

you to provide error reporting specific to your particular problem. For instance:

```
class bad_packet_error : public runtime_error
{
   public:
      bad_packet_error(const string& what);
};

if (bad_packet())
   throw bad_packet_error("Packet size incorrect");
```

This demonstrates how the Standard C++ exception classes provide you with a basic error model. From this foundation you can build the right error detection and reporting methods required for your particular application.

18.4 Example Program

This following example program demonstrates the use of exceptions.

Obtaining the Sample Program.
This program can be found in the file `exceptn.cpp` *in your code distribution.*

```
#include <stdexcept>
#include <string>

static void f() { throw runtime_error("a runtime error"); }

int main ()
{
  string s;

  // First we'll try to incite then catch an exception from
  // the standard library string class.
  // We'll try to replace at a position that is non-existent.
  //
  // By wrapping the body of main in a try-catch block we can be
  // assured that we'll catch all exceptions in the exception
  // hierarchy. You can simply catch exception as is done below,
  // or you can catch each of the exceptions in which you have an
  // interest.
  try
  {
    s.replace(100,1,1,'c');
  }
  catch (const exception& e)
  {
    cout << "Got an exception: " << e.what() << endl;
  }

  // Now we'll throw our own exception using the function
  // defined above.
  try
  {
    f();
  }
  catch (const exception& e)
  {    cout << "Got an exception: " << e.what() << endl;
  }

 return 0;
}
```

Section 19. auto_ptr

19.1 Overview

The ***auto_ptr*** class wraps any pointer obtained through new and provides automatic deletion of that pointer. The pointer wrapped by an ***auto_ptr*** object is deleted when the ***auto_ptr*** itself is destroyed.

19.1.1 Include File

Include the memory header file to access the ***auto_ptr*** class.

```
#include <memory>
```

19.2 Declaration and Initialization of Auto Pointers

You attach an ***auto_ptr*** object to a pointer either by using one of the constructors for ***auto_ptr***, by assigning one ***auto_ptr*** object to another, or by using the reset member function. Only one ***auto_ptr*** "owns" a particular pointer at any one time, except for the NULL pointer (which all ***auto_ptr***s own by default). Any use of ***auto_ptr***'s copy constructor or assignment operator transfers ownership from one ***auto_ptr*** object to another. For instance, suppose we create ***auto_ptr*** a like this:

```
auto_ptr<string> a(new string);
```

The ***auto_ptr*** object a now "owns" the newly created pointer. When a is destroyed (such as when it goes out of scope) the pointer will be deleted. But, if we assign a to b, using the assignment operator:

```
auto_ptr<string> b = a;
```

b now owns the pointer. Use of the assignment operator causes a to release ownership of the pointer. Now if a goes out of scope the pointer will not be affected. However, the pointer *will* be deleted when b goes out of scope.

The use of new within the constructor for a may seem a little odd. Normally we avoid constructs like this since it puts the responsibility for deletion on a different entity than the one responsible for allocation. But in this case, the ***auto_ptr***'s sole responsibility is to manage the deletion. This syntax is actually preferable since it prevents us from accidentally deleting the pointer ourselves.

Use operator*, operator->, or the member function get() to access the pointer held by an ***auto_ptr***. For instance, we can use any of the three following statements to assign "What's up Doc" to the string now pointed to by the ***auto_ptr*** b.

```
*b = "What's up Doc";
*(b.get()) = "What's up Doc";
 b->assign("What's up Doc");
```

auto_ptr also provides a release member function that releases ownership of a pointer. Any ***auto_ptr*** that does not own a specific pointer is assumed to point to the NULL pointer, so calling release on an ***auto_ptr*** will set it to the NULL pointer. In the example above, when a is assigned to b, the pointer held by a is released and a is set to the NULL pointer.

19.3 Example Program

This program illustrates the use of ***auto_ptr*** to ensure that pointers held in a vector are deleted when they are removed. Often, we might want to hold pointers to strings, since the strings themselves may be quite large and we'll be copying them when we put them into the vector. Particularly in contrast to a string, an ***auto_ptr*** is quite small: hardly bigger than a pointer. (Note that the program runs as is because the vector includes memory.)

Obtaining the Sample Program.
You can find this program in the file autoptr.cpp in the turorial distribution.

```
#include <vector>
#include <memory>
#include <string>

int main()
{
  {
    // First the wrong way
    vector<string*> v;
    v.insert(v.begin(), new string("Florence"));
    v.insert(v.begin(), new string("Milan"));
    v.insert(v.begin(), new string("Venice"));

    // Now remove the first element

    v.erase(v.begin());

  // Whoops, memory leak
    // string("Venice") was removed, but not deleted
    // We were supposed to handle that ourselves
  }
  {
    // Now the right way
    vector<auto_ptr<string> > v;
    v.insert(v.begin(),
             auto_ptr<string>(new string("Florence")));
    v.insert(v.begin(), auto_ptr<string>(new string("Milan")));
    v.insert(v.begin(), auto_ptr<string>(new string("Venice")));

    // Everything is fine since auto_ptrs transfer ownership of
    // their pointers when copied

    // Now remove the first element
    v.erase(v.begin());
    // Success
    // When auto_ptr(string("Venice")) is erased (and destroyed)
    // string("Venice") is deleted
  }

  return 0;
}
```

Section 20. Complex

20.1 Overview

The class ***complex*** is a template class, used to create objects for representing and manipulating complex numbers. The operations defined on complex numbers allow them to be freely intermixed with the other numeric types available in the C++ language, thereby permitting numeric software to be easily and naturally expressed.

20.1.1 Include Files

Programs that use complex numbers must include the `complex` header file.

```
# include <complex>
```

20.2 Creating and Using Complex Numbers

In the following sections we will describe the operations used to create and manipulate complex numbers.

20.2.1 Declaring Complex Numbers

The template argument is used to define the types associated with the real and imaginary fields. This argument must be one of the floating point number data types available in the C++ language, either `float`, `double`, or `long double`.

There are several constructors associated with the class. A constructor with no arguments initializes both the real and imaginary fields to zero. A constructor with a single argument initializes the real field to the given value, and the imaginary value to zero. A constructor with two arguments initializes both real and imaginary fields. Finally, a copy constructor can be used to initialize a complex number with values derived from another complex number.

```
complex<double> com_one;              // value 0 + 0i
complex<double> com_two(3.14);        // value 3.14 + 0i
complex<double> com_three(1.5, 3.14)  // value 1.5 + 3.14i
complex<double> com_four(com_two);    // value is also 3.14 + 0i
```

A complex number can be assigned the value of another complex number. Since the one-argument constructor is also used for a conversion operator, a complex number can also be assigned the value of a real number. The real field is changed to the right hand side, while the imaginary field is set to zero.

```
com_one = com_three;                  // becomes 1.5 + 3.14i
com_three = 2.17;                     // becomes 2.17 + 0i
```

The function `polar()` can be used to construct a complex number with the given magnitude and phase angle.

```
com_four = polar(5.6, 1.8);
```

The conjugate of a complex number is formed using the function `conj()`. If a complex number represents `x + iy`, then the conjugate is the value `x-iy`.

```
complex<double> com_five = conj(com_four);
```

20.2.2 Accessing Complex Number Values

The member functions `real()` and `imag()` return the real and imaginary fields of a complex number, respectively. These functions can also be invoked as ordinary functions with complex number arguments.

```
   // the following should be the same
cout << com_one.real() << "+" << com_one.imag() << "i" << endl;
cout << real(com_one) << "+" << imag(com_one) << "i" << endl;
```

Functions and Member Functions

Note that, with the exception of the member functions `real()` and `imag()`, most operations on complex numbers are performed using ordinary functions, not member functions.

20.2.3 Arithmetic Operations

The arithmetic operators `+, -, *,` and `/` can be used to perform addition, subtraction, multiplication and division of complex numbers. All four work either with two complex numbers, or with a complex number and a real value. Assignment operators are also defined for all four.

```
cout << com_one + com_two << endl;
cout << com_one - 3.14 << endl;
cout << 2.75 * com_two << endl;
com_one += com_three / 2.0;
```

The unary operators `+` and `-` can also be applied to complex numbers.

20.2.4 Comparing Complex Values

Two complex numbers can be compared for equality or inequality, using the operators `==` and `!=`. Two values are equal if their corresponding fields are equal. Complex numbers are not well-ordered, and thus cannot be compared using any other relational operator.

20.2.5 Stream Input and Output

Complex numbers can be written to an output stream, or read from an input stream, using the normal stream I/O conventions. A value is written in parentheses, either as (u) or (u,v), depending upon whether or not the imaginary value is zero. A value is read as a set of parentheses surrounding two numeric values.

20.2.6 Norm and Absolute Value

The function `norm()` returns the norm of the complex number. This is the sum of the squares of the real and imaginary parts. The function `abs()`

returns the absolute value, which is the square root of the norm. Note that both are ordinary functions that take the complex value as an argument, not member functions.

```
cout << norm(com_two) << endl;
cout << abs(com_two) << endl;
```

The directed phase angle of a complex number is yielded by the function `arg()`.

```
cout << com_four << " in polar coordinates is "
   << arg(com_four) << " and " << norm(com_four) << endl;
```

20.2.7 Trigonometric Functions

The trigonometric functions defined for floating point values (namely, `sin()`, `cos()`, `tan()`, `sinh()`, `cosh()`, and `tanh()`), have all been extended to complex number arguments. Each takes a single complex number as argument and returns a complex number as result.

20.2.8 Transcendental Functions

The transcendental functions `exp()`, `log()`, `log10()` and `sqrt()` have been extended to complex arguments. Each takes a single complex number as argument, and returns a complex number as result.

The standard library defines several variations of the exponential function `pow()`. Versions exist to raise a complex number to an integer power, to raise a complex number to a complex power or to a real power, or to raise a real value to a complex power.

20.3 Example Program – Roots of a Polynomial

The roots of a polynomial $a\ x^2 + b\ x + c = 0$ are given by the formula:

$$x = (-b \pm \sqrt{b^2 - 4ac})/2a$$

Obtaining the Sample Program

This program is found in the file `complx.cpp` in the distribution.

The following program takes as input three double precision numbers, and returns the complex roots as a pair of values.

```
typedef complex<double> dcomplex;

pair<dcomplex, dcomplex> quadratic
      (dcomplex a, dcomplex b, dcomplex c)
         // return the roots of a quadratic equation
{
   dcomplex root = sqrt(b * b - 4.0 * a * c);
   a *= 2.0;
   return make_pair(
      (-b + root)/a,
      (-b - root)/a);
}
```

Section 21. Numeric Limits

21.1 Overview

A new feature of the C++ Standard Library is an organized mechanism for describing the characteristics of the fundamental types provided in the execution environment. In older C and C++ libraries, these characteristics were often described by large collections of symbolic constants. For example, the smallest representable value that could be maintained in a character would be found in the constant named `CHAR_MIN`, while the similar constant for a `short` would be known as `SHRT_MIN`, for a float `FLT_MIN`, and so on.

The template class ***numeric_limits*** provides a new and uniform way of representing this information for all numeric types. Instead of using a different symbolic name for each new data type, the class defines a single static function, named `min()`, which returns the appropriate values. Specializations of this class then provide the exact value for each supported type. The smallest character value is in this fashion yielded as the result of invoking the function `numeric_limits<char>::min()`, while the smallest floating point value is found by invoking `numeric_limits<float>::min()`, and so on.

Solving this problem by using a template class not only greatly reduces the number of symbolic names that need to be defined to describe the operating environment, but it also ensures consistency between the descriptions of the various types.

Two Mechanisms, One Purpose

For reasons of compatibility, the numeric_limits mechanism is used as an addition to the symbolic constants used in older C++ libraries, rather than a strict replacement. Thus both mechanisms will, for the present, exist in parallel. However, as the numeric_limits technique is more uniform and extensible, it should be expected that over time the older symbolic constants will become outmoded.

21.2 Fundamental Data Types

The standard library describes a specific type by providing a specialized implementation of the `numeric_limits` class for the type. Static functions and static constant data members then provide information specific to the type. The standard library includes descriptions of the following fundamental data types.

`bool`	`char`	`int`	`float`
	`signed char`	`short`	`double`
	`unsigned char`	`long`	`long double`
	`wchar_t`	`unsigned short`	
		`unsigned int`	
		`unsigned long`	

Certain implementations may also provide information on other data types. Whether or not an implementation is described can be discovered using the static data member field `is_specialized`. For example, the following is

legal, and will indicate that the ***string*** data type is not described by this mechanism.

```
cout << "are strings described " <<
   numeric_limits<string>::is_specialized << endl;
```

For data types that do not have a specialization, the values yielded by the functions and data fields in `numeric_limits` are generally zero or false.

21.3 Numeric Limit Members

Since a number of the fields in the `numeric_limits` structure are meaningful only for floating point values, it is useful to separate the description of the members into common fields and floating-point specific fields.

21.3.1 Members Common to All Types

The following table summarizes the information available through the ***numeric_limits*** static member data fields and functions.

Type	Name	Meaning
`bool`	`is_specialized`	true if a specialization exists, false otherwise
`T`	`min()`	smallest finite value
`T`	`max()`	largest finite value
`int`	`radix`	the base of the representation
`int`	`digits`	number of `radix` digits that can be represented without change
`int`	`digits10`	number of base-10 digits that can be represented without change
`bool`	`is_signed`	true if the type is signed
`bool`	`is_integer`	true if the type is integer
`bool`	`is_exact`	true if the representation is exact
`bool`	`is_bounded`	true if representation is finite
`bool`	`is_modulo`	true if type is modulo
`bool`	`traps`	true if trapping is implemented for the type

Radix represents the internal base for the representation. For example, most machines use a base 2 radix for integer data values, however some may also support a representation, such as BCD, that uses a different base. The `digits`

field then represents the number of such radix values that can be held in a value. For an integer type, this would be the number of non-sign bits in the representation.

All fundamental types are bounded. However, an implementation might choose to include, for example, an infinite precision integer package that would not be bounded.

A type is *modulo* if the value resulting from the addition of two values can wrap around, that is, be smaller than either argument. The fundamental unsigned integer types are all modulo.

21.3.2 Members Specific to Floating Point Values

The following members are either specific to floating point values, or have a meaning slightly different for floating point values than the one described earlier for non-floating data types.

Type	Name	Meaning
`T`	`min()`	the minimum positive normalized value
`int`	`digits`	the number of digits in the mantissa
`int`	`radix`	the base (or radix) of the exponent representation
`T`	`epsilon()`	the difference between 1 and the least representable value greater than 1
`T`	`round_error()`	a measurement of the rounding error
`int`	`min_exponent`	minimum negative exponent
`int`	`min_exponent10`	minimum value such that 10 raised to that power is in range
`int`	`max_exponent`	maximum positive exponent
`int`	`max_exponent10`	maximum value such that 10 raised to that power is in range
`bool`	`has_infinity`	true if the type has a representation of positive infinity
`T`	`infinity()`	representation of infinity, if available
`bool`	`has_quiet_NaN`	true if there is a representation of a quiet ``Not a Number”
`T`	`quiet_NaN()`	representation of quiet NaN, if available
`bool`	`has_signaling_NaN`	true if there is a representation for

Type	Name	Meaning
		a signaling NaN
`T`	`signaling_NaN()`	representation of signaling NaN, if available
`bool`	`has_denorm`	true if the representation allows denormalized values
`T`	`denorm_min()`	Minimum positive denormalized value
`bool`	`is_iec559`	true if representation adheres to IEC 559 standard.
`bool`	`tinyness_before`	true if `tinyness` is detected before rounding
	`round_style`	rounding style for type

For the `float` data type, the value in field `radix`, which represents the base of the exponential representation, is equivalent to the symbolic constant `FLT_RADIX`.

For the types `float, double` and `long double` the value of `epsilon` is also available as `FLT_EPSILON, DBL_EPSILON`, and `LDBL_EPSILON`.

A NaN is a “Not a Number.” It is a representable value that nevertheless does not correspond to any numeric quantity. Many numeric algorithms manipulate such values.

The IEC 559 standard is a standard approved by the International Electrotechnical Commission. It is the same as the IEEE standard 754.

Value returned by the function `round_style()` is one of the following: `round_indeterminate, round_toward_zero, round_to_nearest, round_toward_infinity`, or `round_toward_neg_infinity.`

Section 22.
Run Time Support

22.1 Overview

The C++ Standard Library headers <new> and <typeinfo> define the interface to the run-time support needed by C++ programs in addition to that provided by the Standard C library. Header <new> provides prototypes for the storage allocation and deallocation operators as well as the exception class bad_alloc used to report storage allocation errors. Header <typeinfo> forms the interface to the new run-time type identification (RTTI) mechanism. It defines the type_info class along with exception classes bad_typeid and bad_cast used to report run-time type violation errors.

22.2 Header <new> synopsis

Programs that intend to use the new storage allocation and deallocation operators or handle storage allocation errors via exceptions must include the <new> header file

```
#include <new>
```

Header <new> contains the following:

```
namespace std {
   class bad_alloc;
   struct nothrow_t { };
   extern const nothrow_t nothrow;
   typedef void (*new_handler)();
   new_handler set_new_handler(new_handler) throw();
}
```

Class ***bad_alloc*** is used to report storage allocation errors. The type nothrow_t and the predefined constant nothrow of this type is used to call the non-exception throwing versions of the allocation functions. Function type new_handler and the function set_new_handler are provided for handling storage allocation errors. In addition the following global operators are provided:

```
1    void*    operator new(size_t) throw(std::bad_alloc)
2    void*    operator new(size_t, const std::nothrow_t&) throw()
3    void*    operator new[](size_t) throw(std::bad_alloc)
4    void*    operator new[](size_t, const std::nothrow_t&) throw()
5    void*    operator new(size_t, void*) throw();
6    void*    operator new[](size_t, void*) throw()
7    void     delete(void*) throw();
8    void     delete(void*, const std::nothrow_t&) throw();
9    void     delete[](void*) throw();
10    void     delete[](void*, const std::nothrow_t&) throw();
```

```
11    void     delete(void*, void*) throw();
12    void     delete[](void*, void*) throw();
```

All of the above function signatures are in the global namespace, i.e., no `using` directive or namespace qualification is necessary. Nos. 1, 3, 7 and 9 will be familiar are the "old" versions of `new` and `delete` except that the `new` allocation functions now throw ***bad_alloc*** exceptions on failure. Any of these functions may be replaced by the user except 11 and 12 (placement `delete`), which have been left unimplemented. Since these cannot be replaced according to the ANSI C++ Draft Standard, their implementation is left up to the user though the Draft Standard says that these intentionally do not perform any action and are only provided to complement Nos. 5 and 6.

22.3 Single-object forms of operators new and delete

```
void*  operator new(size_t size) throw(bad_alloc)
```

Effects: This is the allocation function called by a new expression to allocate `size` bytes of storage suitably aligned to represent any object of that size.

Default behavior:

- Execute a loop: within the loop the function first attempts to allocate the requested storage.
- Return a pointer to the allocated storage if the attempt is successful. Otherwise, if the last argument to `set_new_handler()` was a null pointer, throw `bad_alloc`.
- Otherwise, the function calls the current `new_handler`. If the called function returns, the loop repeats.
- The loop terminates when an attempt to allocate the requested storage is successful or when a called `new_handler` function does not return.

```
void* operator new(size_t, const std::nothrow_t&) throw()
```

Effects: Same as above, except that it is called by a placement version of a new-expression when a C++ program wants a null pointer result as an error indication, instead of a `bad_alloc` exception.

Default behavior:

- Execute a loop: within the loop the function first attempts to allocate the requested storage.
- Return a pointer to the allocated storage if the attempt is successful. Otherwise, if the last argument to `set_new_handler()` was a null pointer, return a null pointer.
- The loop terminates when an attempt to allocate the requested storage is successful or when a called `new_handler` function does not return. If the

called new_handler function terminates by throwing a bad_alloc exception, the function returns a null pointer.

Examples:

```
T* ptr;
try {
    ptr = new T;    // can throw bad_alloc if allocation fails
}
catch(const bad_alloc& err) {
    //...
}
```

If error reporting is to be done via a null pointer then the following code can be used:

```
T* ptr;
ptr = new (nothrow) T;
if (ptr == 0)
    //...
```

```
void operator delete(void* ptr) throw();
void operator delete(void* ptr, const std::nothrow_t&) throw();
```

Effects: The deallocation function is called by a delete expression to render the value of ptr invalid and return the allocated storage back to where it was obtained from.

Default behavior:

- For a null value of ptr do nothing
- Any other value of ptr shall be a value returned earlier by a call to the default operator new which was not invalidated by an intervening call to operator delete(void*). For such a non-null value of ptr, reclaims storage allocated earlier by the call to the default operator new.

Examples:

```
T* ptr = new T;
delete ptr;

T* ptr = new T;
delete (nothrow) ptr;
```

22.4 Array forms of operators new and delete

```
void* operator new[](size_t size) throw(bad_alloc);
```

Effects: The allocation function called by the array form of a new-expression to allocate size bytes of storage suitably aligned to represent any array object of that size or smaller. Note that it is not the direct responsibility of operator new[](size_t) to record the repetition count or element size of the array; that is done elsewhere in the array new- and array delete-expressions. The

array new-expression may however increase the size argument to obtain space for storing supplemental information. The allocation function need only return the exact size of storage requested.

Default behavior: returns `operator new(size)`.

```
void* operator new[](size_t, const std::nothrow_t&) throw();
```

Effects: Same as above except that it is called by a placement version of a new-expression when a C++ program prefers a null pointer result as an error indication, instead of a `bad_alloc` exception.

Default behavior: returns operator new(size, nothrow).

```
void operator delete[](void* ptr) throw();
```

```
void operator delete[](void* ptr, const std::nothrow_t&) throw();
```

Effects: The deallocation function called by the array form of a delete-expression to render the value of ptr invalid.

Default behavior: the same as `operator delete(void*)` and `operator delete(void*, const std::nothrow_t&)` except that `ptr` should have been obtained from a prior call to `operator new[]` and not been invalidated by an intervening call to `operator delete[](void*)`.

Examples:

```
T* ptr;
try {
  ptr = new T[100];     // array new-expression
  delete[] ptr;         // array delete-expression
}
catch (const bad_alloc& err) {
    //...
}

T* ptr = new (nothrow) T[100];    // nothrow version
if (ptr)
    //...
delete (nothrow)[] ptr;
```

Note that in the above example, the new-expression may ask for more than 100*sizeof(T) bytes of storage, but the called allocation function returns only what is requested.

22.5 Placement forms of operators new and delete

These functions are reserved: a C++ program may not define functions that displace the versions in the standard C++ library.

```
void* operator new(size_t size, void* ptr) throw();
```

```
void* operator new[](size_t size, void* ptr) throw();
```

Effects: both return `ptr` and perform no other action.

Example: The placement new function is useful for construction of an object at a known address:

```
long mem[128];     // long is usually the most restrictive type
AThing* ptr;       // assume size of(AThing) <= 128*sizeof(long)

ptr = new (&mem[0]) AThing;       // use mem[] as storage
                                     or, if mem[] is big enough:
ptr = new (&mem[0]) AThing[8];    // allocate array[8] of Athing
                                     starting at mem[0]
```

```
void delete(void* ptr, void*) throw();
```

```
void delete[](void* ptr, void*) throw();
```

Effects: Performs no action according to the ANSI C++ Draft Standard. However, in this implementation of the standard, these are left unimplemented. Ordinarily, there should be no need for using these two functions.

22.6 Storage allocation errors

Storage allocation errors in allocation functions are handled via two mechanisms: throwing exceptions and calling handler functions. These methods may used together or exclusively depending on the allocation function being called (22.3, 22.4) and the current handler function (22.6.2, 22.6.3).

22.6.1 Class bad_alloc

```
#include <exception>

namespace std {

    class bad_alloc : public exception {
    public:
        bad_alloc() throw();
        bad_alloc(const bad_alloc&) throw();
        bad_alloc& operator=(const bad_alloc&) throw();
        virtual ~bad_alloc() throw();
        virtual const char* what() const throw();
    };
}
```

The class ***bad_alloc*** defines the type of objects thrown as exceptions by the implementation to report a failure to obtain storage.

The member function `what()` returns the string `"bad_alloc"`.

22.6.2 Type new_handler

```
typedef void (*new_handler)();
```

The type of a (possibly user supplied) handler function used when the allocation functions cannot satisfy a request for additional storage. A user supplied new handler has the following behavior required of it:

- make more storage available for allocation and then return;
- otherwise throw an exception of type `bad_alloc` or a class derived from `bad_alloc`
- or else call either `abort()` or `exit()`

Default behavior: throws an exception of type `bad_alloc`.

22.6.3 set_new_handler

```
new_handler set_new_handler(new_handler new_p) throw();
```

Effects: Establishes the function designated by `new_p` as the current handler function.
Returns: the previous handler function.

Example:

```
#include <new>

void out_of_memory()
{
   int status = expand_storage();  // some system function
   if (status == FAILED)throw bad_alloc();  // or call abort() or exit()
   else return;  // success
}

//...

void f()
{
   new_handler prev_new_handler = set_new_handler(out_of_memory);
   T* ptr;

   try {
      ptr = new T;  // out_of_memory may be called on failure
   }
   catch(...) {
   }
   // ...
   set_new_handler(prev_new_handler);  // restore old handler
}
```

In general, if the handler function throws an exception on failure to obtain more storage, the call to `new` must be wrapped in a try block and a catch handler defined for the exception, unless a nothrow version of new is being used, which catches any exceptions thrown from the handler function internally and just returns a null pointer.

22.7 Run-time type identification (RTTI)

C++ implements RTTI via the operator `typeid` and the class `type_info` defined in the header `<typeinfo>`. The result of applying the operator `typeid`

to an expression or a type name is an object of type `std::type_info`. An object of this type cannot be constructed explicitly; the only way to do this is to use the `typeid` operator.

The synopsis of header `<typeinfo>` is

```
namespace std {
    class type_info;
    class bad_cast;
    class bad_typeid;
}
```

22.7.1 class type_info

```
namespace std {
    class type_info {
    public:
        virtual ~type_info();
        bool operator==(const type_info&) const;
        bool operator!=(const type_info&) const;
        bool before(const type_info&) const;
        const char* name() const;
    };
}
```

The class ***type_info*** describes type of the object representing information generated by the `typeid` operator. The `name()` member function returns a pointer to a null terminated string representing the C++-style type name of the object. The operators == and != provide means to compare type information from different types or objects. The member function `before(const type_info&)` provides means for collating type_information, such as may be needed for storing it in an ordered container such as a binary tree. This implementation does not do a lexicographical comparison of the strings returned by `name()`; instead an internally generated address comparison is used. RTTI can be used for implementing persistent objects.

22.7.2 class bad_cast

```
#include <exception>

namespace std {
    class bad_cast : public exception {
    public:
        bad_cast() throw();
        bad_cast(const bad_cast&) throw();
        bad_cast& operator=(const bad_cast&) throw();
        virtual ~bad_cast() throw();
        virtual const char* what() const throw();
    };
}
```

The class ***bad_cast*** defines the type of the objects thrown as exceptions by the implementation to report the execution of an invalid dynamic-cast expression. The member function `what()` returns the string `"bad_cast"` for identification.

Example:

```
void f(base_class& base_ref)
{
    derived_class& ref;
    try {
        ref = dynamic_cast<derived_class&>(base_ref);
        // OK; use ref now
    }
    catch (const bad_cast& cast_err) {
        // base_ref was not a ref to derived_class
    }
}
```

The `dynamic_cast` operator together with the class `bad_cast` provides more robustness when using polymorphism than was previously available.

22.7.3 class bad_typeid

```
#include<exception>

namespace std {
    class bad_typeid : public exception {
    public:
        bad_typeid() throw();
        bad_typeid(const bad_typeid&) throw();
        bad_typeid& operator=(const bad_typeid&) throw();
        virtual ~bad_typeid() throw();
        virtual const char* what() const throw();
    };
}
```

The class ***bad_typeid*** defines the type of objects thrown as exceptions by the implementation to report null pointer in a `typeid` expression. The member function `what()` returns the string `"bad_typeid"`.

23. Glossary

bidirectional iterator An iterator that can be used for reading and writing, and which can move in either a forward or backward direction.

binary function A function that requires two arguments.

binder A function adaptor that is used to convert a two-argument binary function object into a one-argument unary function object, by binding one of the argument values to a specific constant.

constant iterator An iterator that can be used only for reading values, which cannot be used to modify the values in a sequence.

container class A class used to hold a collection of similarly typed values. The container classes provided by the standard library include vector, list, deque, set, map, stack, queue, and priority_queue.

deque An indexable container class. Elements can be accessed by their position in the container. Provides fast random access to elements. Additions to either the front or the back of a deque are efficient. Insertions into the middle are not efficient.

forward iterator An iterator that can be used either for reading or writing, but which moves only forward through a collection.

function object An instance of a class that defines the parenthesis operator as one of its member functions. When a function object is used in place of a function, the parenthesis member function will be executed when the function would normally be invoked.

generic algorithm A templated algorithm that is not specialized to any specific container type. Because of this, generic algorithms can be used with a wide variety of different forms of container.

heap A way of organizing a collection so as to permit rapid insertion of new values, and rapid access to and removal of the largest value of the collection.

heterogeneous collection A collection of values that are not all of the same type. In the standard library a heterogeneous collection can only be maintained by storing pointers to objects, rather than objects themselves.

insert iterator An adaptor used to convert iterator write operations into insertions into a container.

iterator A generalization of the idea of a pointer. An iterator denotes a specific element in a container, and can be used to cycle through the elements being held by a container.

generator A function that can potentially return a different value each time it is invoked. A random number generator is one example.

input iterator An iterator that can be used to read values in sequence, but cannot be used for writing.

list A linear container class. Elements are maintained in sequence. Provides fast access only to the first and last elements. Insertions into the middle of a list are efficient.

map An indexed and ordered container class. Unlike a vector or deque, the index values for a map can be any ordered data type (such as a string or character). Values are maintained in sequence, and can be efficiently inserted, accessed or removed in any order.

multimap A form of map that permits multiple elements to be indexed using the same value.

multiset A form of set that permits multiple instances of the same value to be maintained in the collection.

negator An adaptor that converts a predicate function object, producing a new function object that when invoked yields the opposite value.

ordered collection A collection in which all values are ordered according to some binary comparison operator. The set data type automatically maintains an ordered collection. Other collections (vector, deque, list) can be converted into an ordered collection.

output iterator An iterator that can be used only to write elements into a container, it cannot be used to read values.

past the end iterator An iterator that marks the end of a range of values, such as the end of the set of values maintained by a container.

predicate A function or function object that when invoked returns a boolean (true / false) value or an integer value.

predicate function A predicate.

priority_queue An adaptor container class, usually built on top of a vector or deque. The priority queue is designed for rapidly accessing and removing the largest element in the collection.

queue An adaptor container class, usually built on top of a list or deque. The queue provides rapid access to the topmost element. Elements are removed from a queue in the same order they are inserted into the queue.

random access iterator An iterator that can be subscripted, so as to access the values in a container in any order.

range A subset of the elements held by a container. A range is typically specified by two iterators.

reverse iterator An iterator that moves over a sequence of values in reverse order, such as back to front.

sequence A portion or all of the elements held by a container. A sequence is usually described by a range.

set A ordered container class. The set container is optimized for insertions, removals, and tests for inclusion.

stack An adaptor container class, built usually on top of a vector or deque. The stack provides rapid access to the topmost element. Elements are removed from a stack in the reverse of the order they are inserted into the stack.

stream iterator An adaptor that converts iterator operations into stream operations. Can be use to either read from or write to an iostream.

unary function A function that requires only one argument. Applying a binder to a binary function results in a unary function.

vector An indexable container class. Elements are accessed using a key that represents their position in the container. Provides fast random access to elements. Addition to the end of a vector is efficient. Insertion into the middle is not efficient.

wide string A string with 16-bit characters. Wide strings are necessary for many non-roman alphabets, i.e., Japanese.

24. Index