Chapter 1

Introduction

The Standard Template Library—STL—provides a set of C++ container classes and template algorithms designed to work together to produce a wide range of useful functionality. Though only a small number of container classes are provided, they include the most widely useful containers, such as vectors, lists, sets, and maps. The template algorithm components include a broad range of fundamental algorithms for the most common kinds of data manipulations, such as searching, sorting, and merging.

The critical difference between STL and all other C++ container class libraries is that STL algorithms are *generic*: every algorithm works on a variety of containers, *including built-in types*, and many work on *all* containers. In Part I of this book we look at the why and how of generic algorithms and other key concepts that give STL many advantages over other software libraries. One of the most important concepts of STL is the way generic algorithms are defined in terms of *iterators*, which generalize C/C++ pointers, together with the way different kinds of iterators are defined for traversing the different kinds of containers. Besides containers, generic algorithms, and iterators, STL also provides *function objects*, which generalize ordinary C/C++ functions and allow other components to be efficiently adapted to a variety of tasks. The library also includes various other kinds of *adaptors*, for changing the interfaces of containers, iterators, or function objects, and *allocators*, for controlling storage management. All of these components are discussed in the STL overview in Chapter 2 and in more detail in later chapters of Part I.

Just reading about STL may be interesting, but to become really proficient in *using* the library, you'll have to get some actual programming experience with it. Our descriptions in Part I include many small examples that show how individual components work, and Part II presents and explains a series of more substantial programs. Though still small, these examples perform some nontrivial and useful tasks, displaying some of the power that a good software library makes available. Part III contains a complete reference guide to the library.

STL is only one part of a larger software library, the C++ Standard Library approved by the ANSI/ISO C++ committee in its X3J16 report (Ref. 1).[1] Nevertheless, STL remains a coherent framework of fine-grained, interchangeable components that deserve treatment separate from the rest of the C++ Standard Library. In this book we attempt to provide a complete and precise

[1] See the Preface for more information on the background of STL and how it came to be included in the C++ Standard.

user-level description of STL. (For a thorough description of one widely-used implementation of STL, see Ref. 10.)

1.1 Who Should Read This Book

If you are not already familiar with any other software libraries, that shouldn't stop you from reading this book. Although comparisons to other libraries are made in a few places, the main points should be understandable without that background. All that's assumed is that you have some experience with the major C++ concepts: functions, classes, objects, pointers, templates, and stream input/output. Many books on C++ provide the needed background. Crucial features of templates with which some readers might not be familiar (and which might not be covered yet in some books, as these features have only recently been added to the language standard) are described in Section 1.3.

1.2 What Generic Programming Is and Why It's Important

STL is the embodiment of years of research on generic programming. The purpose of this research has been to explore methods of developing and organizing libraries of generic—or reusable—software components. Here the meaning of "reusable" is, roughly, "widely adaptable, but still efficient," where the adaptation is done by preprocessor or programming-language mechanisms rather than manual text editing. There has been a substantial amount of other work in software reuse (other terms often used in this connection are "software building blocks" or "software ICs"), but two distinguishing characteristics of the work that led to STL are the high degree of *adaptability* and *efficiency* of the components.

The essential ideas of the generic component construction approach are shown in the depiction in **Figure 1-1** of library components and the way they "plug together." On the right are components called *generic algorithms*, for operations such as sequence merging, sorting, copying, etc. But these algorithms are not self-contained; they are written in terms of *container access* operations, which are assumed to be provided externally. Providing these container access operations is the role of the components called *iterators* (which are depicted in **Figure 1-1** as "ribbon cables"). Each kind of iterator defines container access operations for a particular data representation, such as a linked-list representation of sequences, or an array representation.

For this fine-grained component approach to work well, there must be a certain minimal level of support from the base programming language. Fortunately, C++ provides such support, especially with its *template* mechanism. Programmers who want to use one of the generic algorithms need only select the algorithm and the container with which it is to be used. The C++ compiler takes care of the final step of plugging together the algorithm, which is expressed as a template function, with the iterators, which are classes that are associated with the container class.

Programmers can write their own iterator class definitions, perhaps in terms of some particular list node structure already in use in existing code. An important point is that it's generally a lot easier to program such an iterator definition than it would be to recode all the generic algorithms to work with that existing data structure.

Not every generic algorithm and iterator pair can be plugged together; you could say that they must be "plug-compatible." (The enforcement of this in C++ is by means of the normal type-checking C++ compilers do when templates are instantiated. Attempts to combine incompatible components result

in compile-time errors.) While it would be *possible* to make all pairs interface together, it turns out to be better not to, because some algorithms are efficient only when using a particular data representation. For example, a sort algorithm may be efficient only for a random access data structure like an array, not for a linked list. In such a case the library should provide a separate algorithm that is suitable for use with lists.

The software library designs that have resulted from this generic programming approach are markedly different from other software libraries:

- The precisely-organized, interchangeable building blocks that result from this approach permit many more useful combinations than are possible with more traditional component designs.
- The design is also a suitable basis for further development of components for specialized areas such as databases, user interfaces, and so on.
- By employing compile-time mechanisms and paying due regard to algorithm issues, component generality can be achieved without sacrificing efficiency. This is in sharp contrast to the inefficiencies often introduced by other C++ library structures involving complex inheritance hierarchies and extensive use of virtual functions.

The bottom-line result of these differences is that generic components are far more *useful* to programmers, and therefore far more likely to be *used,* in preference to programming every algorithmic or data structure operation from scratch.

These are tall promises, and you may well be skeptical that STL, or any library, can fully live up to them. As you read on, though, and start putting this remarkable library to use in your own programming, we believe you will agree that STL truly does fulfill the promise of generic programming.

1.3 How C++ Templates Enable Generic Programming

Adaptability of components is essential to generic programming, so now let's look at how C++ templates enable it. There are two kinds of templates in C++: *template classes* and *template functions.*

1.3.1 Template Classes

Template classes have many uses, but the most obvious one is to provide adaptable storage containers. To take a very simple example, suppose we want to create objects that can store two values, an integer and a character. For this purpose we could define

```
class pair_int_char {
 public:
  int first;
  char second;
  pair_int_char(int x, char y) : first(x), second(y) { }
};
```

We could then write, for example,

```
pair_int_char pair1(13, 'a');
cout << pair1.first << endl;
cout << pair1.second << endl;
```

If we also want objects that can store, say, a boolean value and a double-precision floating-point number, we could define


```
class pair_bool_double {
 public:
  bool first;
  double second;
  pair_bool_double(bool x, double y) : first(x), second(y){}
};
```

and write, for example,

```
pair_bool_double pair2(true, 0.1);
cout << pair2.first << endl;
cout << pair2.second << endl;
```

The same could be repeated for any of the other infinitely many pairs of types, but a template class permits them all to be expressed with a single definition:

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

Here we've written the class definition in terms of two arbitrary type names, `T1` and `T2`, which are *type parameters*. The names are introduced into the definition as type parameters by the clause

```
template <class T1, class T2>
```

which means that we can substitute *actual types* for `T1` and `T2` to obtain a particular *specialization* of the `pair` template class. For example, writing

```
pair<int, char> pair3(13, 'a');
pair<bool, double> pair4(true, 0.1);
```

declares `pair3` and `pair4` with structure equivalent to `pair1` and `pair2`, respectively. But now we can also use `pair` in countless other ways with other combinations of actual types, such as

```
pair<double, long> pair5(3.1415, 999);
pair<bool, bool> pair6(false, true);
```

Types defined by other classes can be used as the actual types; for instance,

```
pair<pair_int_char, float> pair7(pair1, 1.23);
```

or, equivalently,

```
pair<pair<int, char>, float> pair8(pair3, 1.23);
```

As illustrated here, we can use types produced as template class specializations anywhere that ordinary types can be used.

The template class definition of `pair` is a very simple example of a *generic container class* since it is adaptable to many different uses. However, there is usually more to making an ordinary class into a generic one (in the sense of being truly useful in the widest range of applications) than just adding the `template` keyword and type parameter list. Some modifications may be necessary to ensure that the resulting instances are as efficient as ones defined by ordinary means, no matter what actual types are substituted. Even in the case of a template class definition as simple as `pair`, an improvement can be made. In defining the template class, we wrote the constructor declaration a little too simply:

```
pair(T1 x, T2 y) : first(x), second(y) { }
```

The problem with this is that `x` and `y` are passed *by value*, which means that if they are large objects, an expensive extra copy will be made when the constructor is called. Instead, `x` and `y` should be passed as *constant reference parameters*, so that only an address is passed. And this is, in fact, the way that `pair` is defined as an STL component:

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

One other thing added here is the default constructor, `pair()`, which is useful in situations where a default initialization of `pair` objects is needed in another class definition. In any class definition in which no other constructor is defined, the compiler defines the default constructor for us, but in this case we have defined another constructor and the language rules require us to define the default constructor explicitly.

In the case of more-complex class definitions, there are usually many other factors involved in making them widely adaptable and efficient. In this book we will not examine in depth the way STL is implemented, but STL's originators gave careful consideration to maintaining efficiency while making components general.

1.3.2 Template Functions

Template functions can be used to define generic algorithms. A simple example is a function for computing the maximum of two integers:

```
int intmax(int x, int y)
{
  if x < y
    return y;
  else
    return x;
}
```

This can be used only to compute the maximum of two `int`s, but we can easily generalize it by making it a template function:

```
template <class T>
T max(T x, T y)
{
  if x < y
    return y;
  else
    return x;
}
```

A big difference from template classes is that you do not have to tell the compiler which types you are using; the compiler is able to infer them from the types of the arguments.

```
int u = 3, v = 4;
double d = 4.7;
cout << max(u, v) << endl;   // type int is inferred
cout << max(d, 9.3) << endl; // type double is inferred
```

The compiler requires that values of the same type be passed for `x` and `y` since the same template type parameter, `T`, is used for both in the declaration. Thus the following is an error:

```
cout << max(u, d) << endl;
```

It also requires there to be a definition of the < operator taking two parameters of type T. For example, recalling one of the uses of the pair template class definitions given in the previous section

```
pair<double, long> pair5(3.1415, 999);
```

the following will not compile because there is no definition of operator< on two pair<double, long> objects:

```
return max(pair5, pair5);
```

But it *would* compile if we first defined some meaning for operator< on objects of this type, such as

```
bool operator<(const pair<double, long>& x,
               const pair<double, long>& y)
  // Compare x and y on their first members:
{
  return x.first < y.first;
}
```

The way the compiler is able to infer types and match up operator or function calls with the proper overloaded definitions makes template functions an extremely useful feature for generic programming. We'll see more evidence of this later, especially in Section 2.2 when we examine how STL generic algorithms are defined by template functions.

Before we leave the max example, though, we should note that the definition of this template function can be improved just as we discussed for the constructor in the pair template class definition, by using constant reference parameters:

```
template <class T>
const T& max(const T& x, const T& y) {
  if x < y
    return y;
  else
    return x;
}
```

This is essentially the definition used in STL.

1.3.3 Member Function Templates

In ordinary class definitions or template class definitions, member functions may have template parameters (in addition to any template parameters at the class level). This relatively new C++ feature is used in the template classes defining STL containers. For example, each STL container class has an insert member function that has a template parameter to specify the type of iterators to be used:

```
template <class T>
class vector {
  ...
  template <class InputIterator>
  insert(iterator position, InputIterator first, InputIterator last);
  ...
};
```

As we'll see in Section 6.1, this insert member function can be used to insert elements copied from some other container (such as a list), using iterators supplied by the other container. Defining such a function as a template member function, with the iterator type as a template parameter, makes it much more useful than if the iterator type were a fixed type.

1.3.4 Default Template Parameters

Another relatively new C++ template feature is default template parameters. For example, the actual form of the STL `vector` template class definition is

```
template <class T, class Allocator = allocator<T> >
class vector {
  ...
};
```

The second template parameter, `Allocator`, has a default value, given by `allocator`. All STL container classes use storage management facilities as provided by `Allocator`, and `allocator` is a type (defined by a class) that provides a standard form of these facilities. Since `Allocator` has a default value, we do not have to pass any allocator when we create an instance of `vector`; that is, `vector<int>` is equivalent to `vector<int, allocator<int> >`.

Allocators are discussed again briefly in Section 2.6 and are covered in detail in Chapter 22.

1.3.5 The "Code Bloat" Problem with Templates

When different specializations of template classes or template functions are used in the same program, the compiler in effect creates different versions of the source code and compiles each one into executable code. (Actually, different versions aren't usually produced at the source-code level but rather at some intermediate level of code representation.) The main benefit is that each copy is specialized for the types used and thus can be just as efficient as if the specialized code had been written directly. But there is also a potentially severe drawback: if many different instances are used, the many copies can make the executable file huge. This "code bloat" problem is exacerbated by the generality of template classes and functions in a library such as STL, since they are *intended* to have many useful specializations and since programmers are encouraged to use them as frequently as possible.

Fortunately, there are techniques for avoiding the most severe consequences of the code bloat problem, at some (usually minor) cost in efficiency. We discuss these techniques in Sections 11.3 (on page 187) and 17.2 (on page 242).

1.4 Understanding STL's Performance Guarantees

STL is unusual among software libraries in that performance guarantees are included in the interface requirements of all of its components. Such guarantees are in fact crucial to making wise choices among components, such as choosing whether to use a list, vector, or deque to represent a sequence in a particular application. The component performance characteristics are stated using "big-Oh" notation.

1.4.1 The Big-Oh Notation and Related Definitions

In most cases, talking about the computing time of an algorithm is simplified by classifying all inputs to the algorithm into subsets characterized by some simple parameter(s). For algorithms that work on containers, usually the size n of the container is a convenient choice of parameter. For inputs of size N, we then consider the *maximum time*, $T(N)$, the algorithm takes. $T(N)$ is also called the *worst-case* time for inputs of size N.)

To simplify matters further, we focus on how $T(N)$ behaves for large N, and instead of trying to write a precise formula for T(N), we just look for a simple

function that provides an *upper bound* on the function. For example, we might have

$$T(N) \leq cN$$

for some constant c and all sufficiently large N. We say in this case that "the time grows at worst linearly with the size of the input." The constant c might have to be chosen differently when the algorithm is compiled and run on different machines, so one last simplification that is frequently made is to express such bounds in a way that hides the constant.

This is done using *big-Oh* notation; for example, the above relation is written

$$T(N) = O(N)$$

In general, if there is some function f such that

$$T(N) \leq cf(N)$$

for some constant c and all sufficiently large N, we write

$$T(N) = O(f(N))$$

In this book the five main cases of this notation are as follows:

1. $T(N) = O(N)$

   This is the case just mentioned. The algorithm is said to have a *linear time bound* or, more simply, to be *linear time* or just *linear*.

2. $T(N) = O(N^2)$

   The algorithm is said to have a *quadratic time bound*, or to be *quadratic time* or just *quadratic*.

3. $T(N) = O(\log N)$

   The algorithm is said to have a *logarithmic time bound*, or to be *logarithmic time* or just *logarithmic*.

4. $T(N) = O(N \log N)$

   The algorithm is said to have an $N \log N$ *time bound*.

5. $T(N) = O(1)$

   The algorithm is said to have an $O(1)$ *time bound* or *a constant time bound* or to be *constant time*. Note that in this case the big-Oh notation is shorthand for $T(N) \leq c$ for some constant c and all sufficiently large N.

In each of these cases, it is important to keep in mind that we are characterizing computing times with upper bounds on *worst-case computing times*. This can be misleading if the worst case occurs extremely rarely, as is the case, for example, with the quicksort algorithm for sorting sequences in place. The worst-case time for this algorithm is quadratic, which is much slower than other sorting algorithms such as heapsort or mergesort. Yet for most situations the quicksort algorithm is the best choice, since it is faster *on the average* than the other algorithms. The inputs that cause worst case for *sort* are in fact so rare that in most situations they can be ignored. In a few cases such as *sort*, the descriptions we give of computing times go beyond the simple big-Oh bounds on worst-case times to discuss the average times. The average time for an algorithm on inputs of size N is usually calculated by assuming all inputs of size N occur with equal probability. However, other probability distributions might be more appropriate in some situations.

1.4.2 Amortized Time Complexity

In several cases, the most useful characterization of an algorithm's computing time is neither worst-case time nor average time, but *amortized time*. This notion is similar to the manufacturing accounting practice of amortization, in which a one-time cost (such as design cost) is divided by the number of units produced and then attributed to each unit. Amortized time can be a useful way to describe the time an operation on some container takes in cases *where the time can vary widely* as a sequence of the operations is done, but the total time for a sequence of N operations has a *better bound* than just N times the worst-case time.

For example, the worst-case time for inserting at the end of an STL vector is $O(N)$, where N is the size of the container, because if there is no extra room the insert operation must allocate new storage and move all the existing elements into the new storage. However, whenever a vector of length N needs to be expanded, $2N$ spaces are allocated, so no such reallocation and copying are necessary for the next $N-1$ insertions. Each of these next $N-1$ insertions can be done in constant time, for a total of $O(N)$ time for the N insertions, which is $O(1)$ time for the insertion operation when averaged over the N operations. Thus, we say that the *amortized time* for insertion is constant, or that the time is *amortized constant*. Thus, for the above example, the amortized constant time bound for insertion more accurately reflects the true cost than does the linear time bound for the worst case. In general, the amortized time for an operation is the total time for a sequence of N operations divided by N. Note that although amortized time is an average, there is no notion of probability involved, as there is with average computing time.

1.4.3 Limitations of the Big-Oh Notation

The big-Oh notation itself has well-known limitations. The time for an $O(N)$ algorithm might grow at a rate slower than linear growth since it is only an upper bound, or it might grow faster than a linear rate for small N, since the bound only holds for large N. Two $O(N)$ algorithms can differ dramatically in their computing times. One could be uniformly 2, or 10, or 100 times faster than the other, but the big-Oh notation suppresses all such differences. As one moves from one compiler or hardware architecture to another, the hidden constants can change, perhaps enough to alter the case for choosing one algorithm over another.

For these and many other reasons, it is advisable to do empirical performance testing of programs under situations that approximate, to the extent possible, the environment and data that are likely to be encountered in production runs.

1.5 STL Header Files

In order to use STL components in your programs, you must use the preprocessor `#include` directive to include one or more header files. In all of the example programs in this book, we assume that the STL header files are organized and named as in the C++ Standard. Headers in the Standard do not use the `.h` extension for C++ library headers, except for those that are also C library headers. Other differences from earlier organizations of the STL library are the following:

1. The `vector`, `list`, and `deque` container classes are in `<vector>`, `<list>`, and `<deque>`, respectively.

2. Both `set` and `multiset` are in `<set>`, and both `map` and `multimap` are in `<map>`.

3. The `stack` adaptor is in `<stack>`, and the `queue` and `priority_queue` adaptors are in `<queue>`.
4. All STL generic algorithms are in `<algorithm>`, except generalized numeric algorithms, which are in `<numeric>`.
5. STL function objects and function adaptors are in `<functional>`.

For all of the examples in this book we assume the library follows the above conventions. Some compilers may still use the older library organization used in the original Hewlett Packard implementation:

1. The STL container class named *c* is in `<c.h>`; e.g., `vector` is in `<vector.h>`, `list` in `<list.h>`, and so forth.
2. STL container adaptors (`stack`, `queue`, and `priority_queue`) are in `<stack.h>`.
3. All STL generic algorithms are in `<algo.h>`.
4. STL stream iterator classes and iterator adaptors are in `<iterator.h>`.
5. STL function objects and function adaptors are in `<function.h>`.

1.6 Conventions Used in Examples

In this book all major and most minor points are illustrated with actual code examples. The examples in Part I are simple enough to be understandable by sight, without being run. To make it clear to the reader what these programs are supposed to do, we make extensive use of the standard C/C++ `assert` macro, from C header `assert.h`. This macro takes a boolean-valued expression as its single argument and does nothing if that expression evaluates to true, but prints an informative message and terminates the program if the expression evaluates to false. First let's look at an example that doesn't use STL at all.

"ex01-01.cpp" ? ≡

```
// Illustrating the assert macro.
#include <iostream>
#include <assert.h>
#include <string.h>

using namespace std;

int main()
{
  cout << "Illustrating the assert macro." << endl;

  char* string0 = "mark twain";
  char string1[20];
  char string2[20];
  strcpy(string1, string0);
  strcpy(string2, string0);
  int N1 = strlen(string1);
  int N2 = strlen(string2);

  assert(N1 == N2);

  // Put the reverse of string1 in string2:
  for (int k = 0; k != N1; ++k)
    string2[k] = string1[N1-1-k];

  assert(strcmp(string2, "niawt kram") == 0);
  return 0;
}
```

In this example, the first use of `assert` checks that the lengths of `string1` and `string2`, as computed by the standard `strlen` function, are the same. The second use of `assert` checks that `string2` contains the same sequence of characters as the string `"niawt kram"`, as determined by the standard `strcmp` function. If this little program is compiled and run,[2] it merely prints

```
Illustrating the assert macro.
```

But if there is some mistake—either in the computations being illustrated or in how the assertions are written—the execution will be terminated with a message identifying which assertion failed. For example, we might have written the reversal code erroneously as

```
for (int k = 0; k != N1; ++k)
  string2[k] = string1[N1-k];
```

(The error is in writing `string1[N1-k]` instead of `string1[N1-1-k]`.) Then the execution would produce something like[3]

```
Illustrating the assert macro.
Assertion failed: strcmp(string2, "niawt kram") == 0,
  file: c:\stl\examples\ex01-01a.cpp, line: 24
```

We would get a similar message if we wrote the reversal code correctly but wrote the assertion incorrectly; for example,

```
for (int k = 0; k != N1; ++k)
     string2[k] = string1[N1-1-k];

assert(strcmp(string2, "naiwt kram") == 0);
```

where the error is in writing `"naiwt"` instead of `"niawt"`. Of course, there's one other possibility: that we wrote both the code and the assertion incorrectly, but so that they agree. There's no foolproof way to avoid this problem entirely, but one can guard against it to a degree by writing more than one assertion about a computation, with the hope that at least one will catch any errors in the code. That's what we've done in many cases in this book.

1.6.1 An STL Example Written Using the `assert` Macro

We just saw an example of code that reverses a string with a for loop. STL provides a generic algorithm called `reverse` that can reverse many kinds of sequences, including character arrays. Here is an example of its use:

```
"ex01-02.cpp" ? ≡
     // Using the STL generic reverse algorithm with an array.
     #include <iostream>
     #include <algorithm> // for reverse algorithm
     #include <assert.h>
     #include <string.h>
     using namespace std;

     int main()
     {
       cout << "Using reverse algorithm with an array" << endl;
       char string1[] = "mark twain";
       int N1 = strlen(string1);
```

[2] If you want to try compiling and running this or any other example in this book, it is not necessary to type it. Source files for all the examples are available via the Internet (see Appendix B).

[3] In some cases we discuss an example in terms of modifications to a previous example without presenting it in full, as we do here with a modification to `ex01-01.cpp`. The files available on the Internet include complete source files for the modified versions. Two modifications of `ex01-01.cpp` are discussed in this section; they are named `ex01-01a.cpp` and `ex01-01b.cpp`.

```
  reverse(&string1[0], &string1[N1]);
  assert(strcmp(string1, "niawt kram") == 0);
  return 0;
}
```

The arguments to `reverse` are pointers to the beginning and end of the string, where by "end" we mean the first position past the actual string contents. This function reverses the order of the characters in this range in place (unlike our previous piece of code, which put the result in another string).

1.6.2 Examples Using STL Vectors of Characters

In the example above we used simple `char*` strings, which are just arrays of characters. All STL generic algorithms have the nice property that they do work with arrays, but since arrays are a rather weak feature of C/C++, we'll construct most of our examples using one of the sequence containers provided by STL, such as vectors. Vectors provide all the standard features of arrays but are also expandable and have many other useful features, as will be discussed in Sections 2.1.1 and 6.1. Thus, to illustrate `reverse`, a more typical kind of example is the following:

```
"ex01-03.cpp" ? ≡
  // Using the STL generic reverse algorithm with a vector.
  #include <iostream>
  #include <algorithm>
  #include <vector>
  #include <assert.h>
  using namespace std;

  vector<char> vec(char* s)
    // Return vector<char> containing the characters of s
    // (not including the terminating null).
  {
    vector<char> x;
    while (*s != '\0')
      x.push_back(*s++);
    return x;
  }

  int main()
  {
    cout << "Using reverse algorithm with a vector" << endl;
    vector<char> vector1 = vec("mark twain");
    reverse(vector1.begin(), vector1.end());
    assert(vector1 == vec("niawt kram"));
    return 0;
  }
```

This program defines an auxiliary function, `vec`, which makes a character array into a `vector<char>`. The function does this using a member function, `push_back`, to append one character of the string at a time to the vector, which starts out empty. The amount of storage allocated to the vector is expanded as necessary as the characters are appended; this automatic storage allocation is one of the advantages of vectors (and all STL containers) over arrays. The program uses this `vec` function to construct the string to which `reverse` is applied and the string to which to compare the result. In the call of `reverse`, the program uses vector member functions `begin` and `end`, which return the beginning and ending positions of a vector object. Again the convention is that the ending position is the first position past the end of the actual contents.

For the comparison of the result of `reverse` with `vec("niawt kram")`, we use the == operator, since STL defines this operator as an equality test for vectors. In general, STL defines the == operator for all of its container classes. Already we can see one of the advantages of using a vector rather than a bare array: we can write an equality test on character sequences more naturally than with `strcmp`. (The ANSI/ISO Standard C++ library also provides a string class that makes many operations like equality tests more convenient than they are with character arrays. We'll use this string class in later examples.)

1.6.3 Examples Involving User-Defined Types

We use vectors of characters to illustrate STL's generic algorithms mainly because we can easily and concisely construct examples using literal character strings, like `"mark twain"`, by converting them to vectors with simple functions like `vec`. We make similar use of lists of characters, deques of characters, and so on, when we need to illustrate features of these STL containers.

It's important to keep in mind, though, that STL containers can be used to hold objects of any type, not just characters. Any user-defined type `U` defined by a class or struct will also do, although not all container operations will be available unless certain operators are defined on `U`. In particular, many container operations and generic algorithms require the type to have == defined as an equality operator. In order to convey to the reader the essential properties of a type to be used with a particular container operation or generic algorithm, we sometimes construct examples in terms of a class with a minimal definition. For example:

```
"ex01-04.cpp" ? ≡
     // Using the STL generic reverse algorithm with a vector
     // of objects of a minimal type.

     #include <iostream>
     #include <vector>
     #include <algorithm>
     #include <assert.h>
     using namespace std;

     class U {
      public:
       unsigned long id;
       U() : id(0) { }
       U(unsigned long x) : id(x) { }
     };

     // Define == on U objects (needed for vector ==).
     bool operator==(const U& x, const U& y)
     {
       return x.id == y.id;
     }

     int main()
     {
       vector<U> vector1, vector2;
       const int N = 1000;
       int k;
       for (k = 0; k != N; ++k)
         vector1.push_back(U(k));
       for (k = 0; k != N; ++k)
         vector2.push_back(U(N-1-k));
       cout << "Using generic reverse algorithm with a vector "
            << "of user-defined objects" << endl;
```

```
  reverse(vector1.begin(), vector1.end());
  assert(vector1 == vector2);
  return 0;
}
```

Here the class `U` is defined to have a data member `id`. The constructors make it easy to produce an object with a particular `id`. Having such an identifier member isn't necessary to the operation of any of the vector operations or generic algorithms, but it is used here as a convenient way to identify or distinguish objects of the class.

The `reverse` algorithm doesn't use the == operator for type `U`, but we define it on pairs of `U` objects, in terms of the equality of their `id` members, in order to be able to check the equality of two vectors. The == operator is defined on vector objects as long as == is defined on the objects stored in the vectors. For some member functions or generic algorithms, primarily for merging or sorting, we need one other operator, <, defined on objects to be stored in a container. In that case we add a definition similar to that for ==, as follows:

```
bool operator<(const U& x, const U& y)
{
  return x.id < y.id;
}
```

If operator < is defined on the type of objects stored in a container, it is also defined on objects of the container type, in a way described in Chapter 6.

In some cases we may also add

```
ostream& operator<<(ostream& o, const U& x)
{
  o << "U Object #" << x.id;
  return o;
}
```

to output objects the `id` number of a `U` object.

1.6.4 Examples Involving Nested Containers

By giving examples constructed with character sequences or with sequences of objects of a user-defined type like `U`, we show *some* of the generality of STL container operations and generic algorithms, but certainly not all. Another kind of example is sequences in which the objects are other sequences, such as vectors in which the objects are lists of integers:

```
vector<list<int> > vector1, vector2;
```

An example program using container nesting is given in Chapter 14. One important point about such nesting is that if equality is defined on the innermost type parameter, it is defined on the outermost sequences, since each STL container extends its parameter's definition to one for itself. In the preceding example, equality is defined on `int` and is therefore also defined on both `list<int>` and `vector<list<int> >`. Thus, we can write

```
vector1 == vector2
```

to check the equality of `vector1` and `vector2`. The consistent way in which STL provides crucial operations like equality is an important element of its design, as will become apparent as we study STL in more detail.