Chapter 2

Overview of STL Components

STL contains six major kinds of components: *containers*, *generic algorithms*, *iterators*, *function objects*, *adaptors*, and *allocators*. In this chapter we will cover just the highlights of each kind of component, saving the details for later chapters.

2.1 Containers

In STL, containers are objects that store collections of other objects. There are two categories of STL container types: *sequence containers* and *sorted associative containers*.

2.1.1 Sequence Containers

Sequence containers organize a collection of objects, all of the same type `T`, into a strictly linear arrangement. The STL sequence container types are as follows:

- `T a[N]`, that is, ordinary C++ array types, which provide random access to a sequence of fixed length N (*random access* means that the time to reach the ith element of the sequence is constant; that is, the time doesn't depend on i);
- `vector<T>`, providing random access to a sequence of varying length, with constant time insertions and deletions at the end;
- `deque<T>`, also providing random access to a sequence of varying length, with constant time insertions and deletions at both the beginning and the end;
- `list<T>`, providing only linear-time access to a sequence of varying length ($O(N)$, where N is the current length), but with constant time insertions and deletions at *any* position in the sequence.

It may seem surprising that arrays are included in this list, but that's because *all STL generic algorithms are designed to work with arrays in the same way they work with other sequence types.* One example, using STL's generic `reverse` algorithm with strings (character arrays), was given in Section 1.6.1, and we'll

see other examples in Section 2.2. Another case in which many STL algorithms work with standard C++ types is *streams*, as defined by the standard C++ `iostream` library. That is, many algorithms can read from input streams and write their results to output streams.

A container also provides one or more means of stepping through the objects in the collection via other objects called *iterators*, which we'll discuss at length later. For now, let's note that for *all* STL containers—both sequence containers and sorted associative containers—it is possible to step through the objects in the container as though they were arranged in a linear sequence.

Section 1.6.2 gave an example using vectors. The same example can be written using lists instead:

```
"ex02-01.cpp" ? ≡
     // Demonstrating generic reverse algorithm on a list.}

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

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

     int main()
     {
        cout << "Demonstrating generic reverse algorithm on a list"
            << endl;
        list<char> list1 = lst("mark twain");
        reverse(list1.begin(), list1.end());
        assert(list1 == lst("niawt kram"));
        return 0;
     }
```

The example could also be written equally well using deques. As we'll see, vectors, lists, and deques are not completely interchangeable, but in this case each one works as well as the other. That's because each defines `push_back`, `begin`, and `end` member functions with the same abstract meaning, though the implementations are quite different: vectors are represented using arrays; lists are represented using doubly linked nodes; and deques are implemented with a two-level array structure. The only difference that might be apparent to the user in the above example of using the generic `reverse` function would be in *performance*. In this simple case there wouldn't be a noticeable difference in performance, but in other cases, using different algorithms and larger sequences, there can be a tremendous performance advantage of using one kind of sequence over another. (But none is a winner in all cases, which is why more than one is provided in the library.)

2.1.2 Sorted Associative Containers

Sorted associative containers provide an ability for fast retrieval of objects from the collection based on keys. The size of the collection can vary at run time. STL has four sorted associative container types:

- `set<Key>`, which supports unique keys (contains at most one of each key value) and provides for fast retrieval of the keys themselves;
- `multiset<Key>`, which supports duplicate keys (possibly contains multiple copies of the same key value) and provides for fast retrieval of the keys themselves;
- `map<Key, T>`, which supports unique keys (of type `Key`) and provides for fast retrieval of another type `T` based on the keys;
- `multimap<Key, T>`, which supports duplicate keys (of type `Key`) and provides for fast retrieval of another type `T` based on the keys.

A simple example of a sorted associative container is `map<string, long>`, which might be used to hold associations between names and telephone numbers, for example, to represent a telephone directory. Given a name, such a map would provide for fast retrieval of a phone number, as in the following example program.

```
"ex02-02.cpp" ? ≡
     // Demonstrating an STL map
     #include <iostream>
     #include <map>
     #include <string>
     using namespace std;

     int main()
     {
       map<string, long, less<string> > directory;
       directory["Bogart"] = 1234567;
       directory["Bacall"] = 9876543;
       directory["Cagney"] = 3459876;
       // etc.

       // Read some names and look up their numbers.
       string name;
       while (cin >> name)
         if (directory.find(name) != directory.end())
           cout << "The phone number for " << name
                << " is " << directory[name] << "\n";
         else
           cout << "Sorry, no listing for " << name << "\n";
       return 0;
     }
```

In this program, we use the C++ standard library `string` class from the header file `string`. We declare `directory` as a `map` with `string` as the `Key` type and `long` as the associated type `T`. The third template parameter to the map container is `less<string>`, which is a function object used to compare two keys. This particular function object compares strings according to the usual alphabetical ordering (we will discuss such function objects in Section 2.4 and in Chapter 8).

We next insert some names and numbers in the directory with array-like assignments such as `directory["Bogart"] = 1234567`. This notation is possible because the `map` type defines `operator[]` analogously to the corresponding operator on arrays. If we know `name` is in the directory, we can retrieve the associated number with `directory[name]`. In this program we first check to see if `name` is a key stored in directory using `find`, a member function of the `map` container (and all of the sorted associative containers). The `find` function returns an iterator that refers to the entry in the table with `name` as its key if

there is such an entry; otherwise it returns an "off-the-end" iterator, which is the same iterator returned by the `end` member function. Thus, by comparing the iterator returned by `find` with that returned by `end`, we are able to determine whether there is an entry in the table with key `name`.

The STL approach to containers differs in a major way from other C++ container class libraries: STL containers do *not* provide many operations on the data objects they contain. Instead, in STL that's done mainly with *generic algorithms*, the next topic.

2.2 Generic Algorithms

Two of the simplest generic algorithms in STL are `find` and `merge`.

2.2.1 The Generic `find` Algorithm

As a simple example of the flexibility of STL algorithms, consider the `find` algorithm, used to search a sequence for a particular value. It's possible to use `find` with *any* of the STL containers. With arrays, we might write

```
"ex02-03.cpp" ? ≡
     // Demonstrating generic find algorithm with an array.}
     #include <iostream>
     #include <string.h>
     #include <assert.h>
     #include <algorithm>
     using namespace std;

     int main()
     {
       cout << "Demonstrating generic find algorithm with "
            << "an array." << endl;
       const char* s = "C++ is a better C";
       int len = strlen(s);

       // Search for the first occurrence of the letter e.
       const char* where = find(&s[0], &s[len], 'e');

       assert (*where == 'e' && *(where+1) == 't');
       return 0;
     }
```

This program uses `find` to search the elements in `s[0]`, ..., `s[len-1]` to see if any is equal to `e`. If `e` does occur in the array `s`, the pointer `where` is assigned the first position where it occurs, so that `*where == 'e'`. In this case it does occur in the array, but if it didn't, then `find` would return `&s[len]`. This return value is the location one position past the end of the array.

Now, instead of an array, we might have our data stored in a `vector`, a type of container that provides fast random access like arrays but also can grow and shrink dynamically. To find an element in a vector, we can use the *same* `find` algorithm as we used for arrays:

```
"ex02-04.cpp" ? ≡
     // Demonstrating the generic find algorithm with a vector.
     #include <iostream>
     #include <string.h>
     #include <assert.h>
     #include <vector>
     #include <algorithm>
     using namespace std;
```

```
int main()
{
  cout << "Demonstrating generic find algorithm with "
       << "a vector." << endl;

  char* s = "C++ is a better C";
  int len = strlen(s);

  // Initialize vector1 with the contents of string s.
  vector<char> vector1(&s[0], &s[len]);

  // Search for the first occurrence of the letter e.
  vector<char>::iterator
    where = find(vector1.begin(), vector1.end(), 'e');

  assert(*where == 'e' && *(where + 1) == 't');
  return 0;
}
```

This time we construct a vector containing the same characters as array `s`, using a constructor member of class `vector` that initializes the vector using the sequence of values in an array. Instead of `char*`, the type of `where` is `vector<char>::iterator`. *Iterators* are pointer-like objects that can be used to traverse a sequence of objects. When a sequence is stored in a `char` array, the iterators *are* C++ pointers (of type `char*`), but when a sequence is stored in a container such as `vector`, we obtain an appropriate iterator type from the container class. Each STL container type `C` defines `C::iterator` as an iterator type that can be used with type `C` containers.

In either case, when the `find` algorithm is called as in

```
where = find(first, last, value);
```

it assumes that

- iterator `first` marks the position in a sequence where it should *start* processing, and
- `last` marks the position where it can *stop* processing.

Such starting and ending positions are exactly what the `begin` and `end` member functions of the `vector` class (and all other STL classes that define containers) supply.

If the data elements are in a list, once again we can use the same `find` algorithm:

```
"ex02-05.cpp" ? ≡
    // Demonstrating the generic find algorithm with a list.

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

    int main()
    {
      cout << "Demonstrating generic find algorithm with "
           << "a list." << endl;
      char* s = "C++ is a better C";
      int len = strlen(s);
```

```
  // Initialize list1 with the contents of string s.
  list<char> list1(&s[0], &s[len]);

  // Search for the first occurrence of the letter e.
  list<char>::iterator
    where = find(list1.begin(), list1.end(), 'e');

  assert (*where == 'e' && *(++where) == 't');
  return 0;
}
```

There is one subtle difference between this program and the previous one using a vector, due to the fact that the iterators associated with list containers do not support the operator + used in the expression `*(where + 1)`. The reason is explained in Chapter 4. All STL iterators are required to support ++, however, and that is what we use in the expression `*(++where)`.[1]

If we have our data in a `deque`, which is a random-access container similar to arrays and vectors but allowing even more flexibility in the way it can grow and shrink, we can once again use `find`:

"ex02-06.cpp" ? ≡

```
// Demonstrating the generic find algorithm with a deque.
#include <iostream>
#include <string.h>
#include <assert.h>
#include <deque>
#include <algorithm>
using namespace std;

int main()
{
  cout << "Demonstrating generic find algorithm with "
       << "a deque." << endl;
  char* s = "C++ is a better C";
  int len = strlen(s);

  // Initialize deque1 with the contents of string s.
  deque<char> deque1(&s[0], &s[len]);

  // Search for the first occurrence of the letter e.
  deque<char>::iterator
    where = find(deque1.begin(), deque1.end(), 'e');
  assert (*where == 'e' && *(where+1) == 't');
  return 0;
}
```

This program is identical to the vector version except for the substitution of "deque" for "vector" throughout (deque iterators, unlike list iterators, do support the + operator).

In fact, the `find` algorithm can be used to find values in *all* STL containers. The key point with `find` and all other STL generic algorithms is that since they can be used by many or all containers, individual containers do *not* have to define as many separate member functions, resulting in reduced code size and simplified container interfaces.

[1] In analogy to ++ on built-in types, STL defines ++ on iterators to change the value of the iterator as a side-effect, so `++where` is not exactly equivalent to `where+1`. It doesn't matter in this case since the program makes no further use of `where`.

2.2.2 The Generic `merge` Algorithm

The flexibility of STL generic algorithms is even greater than the examples involving `find` have indicated. Consider an algorithm such as `merge`, which combines the elements of two sorted sequences into a single sorted sequence. In general, if `merge` is called as

```
merge(first1, last1, first2, last2, result);
```

it assumes that

- `first1` and `last1` are iterators marking the beginning and end of one input sequence whose elements are of some type `T`;
- `first2` and `last2` are iterators delimiting another input sequence, whose elements are also of type `T`;
- the two input sequences are in ascending order according to the `<` operator for type `T`; and
- `result` marks the beginning of the sequence where the result should be stored.

Under these conditions the result contains all elements of the two input sequences and is also in ascending order. *This interface is flexible enough that the two input sequences and the result sequence can be in different kinds of containers*, as the next example shows.

"ex02-07.cpp" ? ≡

```
// Demonstrating the generic merge algorithm with an array, a
// list, and a deque.
#include <iostream>
#include <string.h>
#include <assert.h>
#include <list>
#include <deque>
#include <algorithm>
using namespace std;

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

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

int main()
{
  cout << "Demonstrating generic merge algorithm with "
       << "an array, a list, and a deque." << endl;
```

```
  char* s = "acegikm";
  int len = strlen(s);
  list<char> list1 = lst("bdfhjlnopqrstuvwxyz");

  // Initialize deque1 with 26 copies of the letter x:
  deque<char> deque1(26, 'x');

  // Merge array s and list1, putting result in deque1:
  merge(&s[0], &s[len], list1.begin(), list1.end(),
        deque1.begin());
  assert(deque1 == deq("abcdefghijklmnopqrstuvwxyz"));
  return 0;
}
```

In this program we create a deque to hold the result of merging array `s` and `list1`. Note that the character sequences in both `s` and `list1` are in ascending order, as is the result produced by `merge` in `deque1`.

We can even merge portions of one sequence with portions of another. For example, we can modify the above program to merge the first 5 characters of `s` with the first 10 characters of `deque1`, putting the result into `list1` (note that we reverse the roles of `list1` and `deque1` from the previous program).

"ex02-08.cpp" ? ≡

```
// Demonstrating generic merge algorithm, merging parts of an
// array and a deque, putting the result into a list.
#include <iostream>
#include <string.h>
#include <assert.h>
#include <list>
#include <deque>
#include <algorithm>
using namespace std;

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

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

int main()
{
  cout << "Demonstrating generic merge algorithm,\n"
       << "merging parts of an array and a deque, putting\n"
       << "the result into a list." << endl;
  char* s = "acegikm";

  deque<char> deque1 = deq("bdfhjlnopqrstuvwxyz");
```

```
  // Initialize list1 with 26 copies of the letter x:
  list<char> list1(26, 'x');

  // Merge first 5 letters in array s with first 10 in
  // deque1, putting result in list1:
  merge(&s[0], &s[5], deque1.begin(), deque1.begin() + 10,
        list1.begin());

  assert(list1 == lst("abcdefghijlnopqxxxxxxxxxxx"));
  return 0;
}
```

These are simple examples, but they already hint at the immense range of possible uses of such generic algorithms.

2.3 Iterators

Understanding iterators is the key to understanding fully the STL framework and learning how to best make use of the library. STL generic algorithms are written in terms of iterator parameters, and STL containers provide iterators that can be plugged into the algorithms, as we saw in **Figure 1-1** in Chapter 1. **Figure 2-1** again depicts this relationship, together with relationships between other major categories of STL components. These very general components are designed to "plug together" in a myriad of different useful ways to produce the kind of larger and more specialized components found in other libraries. The main kind of "wiring" for connecting components together is the category called iterators (drawn as "ribbon cables" in **Figure 2-1** and **Figure 2-2**, which depicts the hierarchical relationship among different iterator categories). One kind of iterator is an ordinary C++ pointer, but iterators other than pointers may exist. These other kinds of iterators are required, however, to behave like pointers in the sense that one can perform operations like `++` and `*` on them and expect them to behave similarly to pointers: for instance, `++i` advances an iterator `i` to the next location, and `*i` returns the location so that it can be stored into, as in `*i = x`, or its value can be used in an expression, as in `x = *i`.

Consider the STL generic function `accumulate`. When called with iterators `first` and `beyond` and a value `init`,

```
accumulate(first, beyond, init);
```

adds up `init` plus the values in positions `first` up to, but not including, `beyond`, and returns the sum. For example, we could write the following program to compute and print the sum of the values in a vector.

```
"ex02-09.cpp" ? ≡
  // Demonstrating the generic accumulate function.
  #include <iostream>
  #include <vector>
  #include <numeric>
  #include <assert.h>
  using namespace std;

  int main()
  {
    cout << "Demonstrating the accumulate function." << endl;
    int x[5] = {2, 3, 5, 7, 11};
    // Initialize vector1 to x[0] through x[4]:
    vector<int> vector1(&x[0], &x[5]);
```

```
  int sum = accumulate(vector1.begin(), vector1.end(), 0);

  assert(sum == 28);
  return 0;
}
```

This program uses `accumulate` to add up the integers in `vector1`, which is traversed using iterators `vector1.begin()` and `vector1.end()`. We could also use `accumulate` with the array `x`, by writing

```
sum = accumulate(&x[0], &x[5], 0);
```

or, say, with a list of `doubles`, as in

```
double y[5] = {2.0, 3.0, 5.0, 7.0, 11.0};
list<double> list1(&y[0], &y[5]);
sum = accumulate(list1.begin(), list1.end(), 0.0);
```

In each case, the abstract meaning is the same—adding to the initial value the values in the range indicated by the iterators—but the *type* of iterators and the *type* of the initial value determine how `accumulate` is adapted to the specific task.

Let's look more closely at the way `accumulate` uses iterators. It can be defined as follows:

```
template <class InputIterator, class T>
T accumulate(InputIterator first, InputIterator beyond, T init)
{
  while (first != beyond)
    init = init + *first++;
  return init;
}
```

The only operations it performs on iterators are incrementing with postfix `++`, dereferencing with `*`, and inequality checking with `!=`. These operations, together with prefix `++` and equality checking, `==`, are the only operations required by the category of iterators called *input iterators*. One other characteristic of input iterators, from which they get their name, is that the `*` operation is only required to be able to read from positions in a container, not to write into them. *Output iterators*, on the other hand, require the same operations except that `*` is required only to be able to write, but not to read.

STL defines three other categories of iterators: *forward* iterators, *bidirectional* iterators, and *random access* iterators. Except for input and output iterators, the relationship between these categories is hierarchical, as shown in **Figure 2-2**. That is, each category adds new requirements to those imposed by the previous category, which means that iterators in a later category are also members of earlier ones. For example, a bidirectional iterator is also a forward iterator, and a random access iterator is also a bidirectional and a forward iterator.

Algorithms that are written to work with input iterators, such as `accumulate`, `find`, and `merge`, are more generic than those that require more powerful iterators, such as `sort`, which requires random access iterators. For example, `sort` cannot be used with STL list containers, because list iterators are only bidirectional and not random access. Instead, STL provides a list member function for sorting that works efficiently with its bidirectional iterators. As we will see in Chapter 4, STL's goal of *efficiency* motivates placing limitations on the generality of some generic algorithms, and the organization of iterators into categories is the chief means of achieving this goal.

2.4 Function Objects

The `accumulate` function discussed in the previous section is very general in terms of its use of iterators, but not as general as it might be in terms of the assumptions it makes about the type of values to which the iterators refer (called the *value type* of the iterators). The `accumulate` definition assumes there is a `+` operator defined on the value type, by its use of `+` in the expression

```
init = init + *first++;
```

This allows the function to be used with any of the C++ built-in numeric types, or with any user-defined type `T` in which such an operator is defined, to add up the values in a sequence. But the abstract notion of accumulation applies to more than just addition; one can equally well accumulate a product of a sequence of values, for example. Thus STL provides another, more general, version of `accumulate`:

```
template <class InputIterator, class T, class BinaryOperation>

T accumulate(InputIterator first, InputIterator last,
             T init, BinaryOperation binary_op)
{
  while (first != last)
    init = binary_op(init, *first++);
  return init;
}
```

Instead of being written in terms of `+`, this definition introduces another parameter, `binary_op`, as the binary operation used to combine values.

How can this more general version of `accumulate` be used to compute a product? If we define a function `mult` as in the following program, we can use it as the `binary_op` parameter to `accumulate`:

"ex02-10.cpp" 0 ≡

```
// Using the generic accumulate algorithm to compute a product,
// using a function object.
#include <iostream>
#include <vector>
#include <numeric>
#include <assert.h>
using namespace std;

class multiply {
 public:
  int operator()(int x, int y) const { return x * y; }
};

int main()
{
  cout << "Using generic accumulate algorithm to "
       << "compute a product." << endl;

  int x[5] = {2, 3, 5, 7, 11};

  // Initialize vector1 to x[0] through x[4]:
  vector<int> vector1(x, x+5);

  int product = accumulate(vector1.begin(), vector1.end(),
                           1, multiply());

  assert(product == 2310);
  return 0;
}
```

(Note that we also changed the initial value from `0` to `1`, which is the proper "identity element" for multiplication.) Here we are passing to `accumulate` an ordinary function, but C++ also supports another possibility: passing a *function object*, by which we mean *an object of a type defined by a class or struct in which the function call operator is defined.* Here's how it can be done, in one of the simplest forms possible:

"ex02-11.cpp" 0 ≡

```
// Using the generic accumulate algorithm to compute a product,
// using a function object.
#include <iostream>
#include <vector>
#include <numeric>
#include <assert.h>
using namespace std;

class multiply {
 public:
  int operator()(int x, int y) const { return x * y; }
};

int main()
{
  cout << "Using generic accumulate algorithm to "
       << "compute a product." << endl;

  int x[5] = {2, 3, 5, 7, 11};

  // Initialize vector1 to x[0] through x[4]:
  vector<int> vector1(x, x+5);

  int product = accumulate(vector1.begin(), vector1.end(),
                           1, multiply());

  assert(product == 2310);
  return 0;
}
```

By defining the function call operator, `operator()`, in class `multiply`, we define a type of object that can be applied to an argument list, just as a function can. Note that the object passed to `accumulate` is obtained by a call of the (default) constructor of the class, `multiply()`. Note also that this object has no storage associated with it, just a function definition (in some cases, though, it is useful to store data in function objects).

What's the advantage, if any, of using function objects rather than ordinary functions? We'll answer this question in detail in Chapter 8, but the main idea is that function objects can carry with them additional information that an ordinary function cannot, and this information can be used by generic algorithms or containers that need more complex knowledge about a function than `accumulate` does. Another important reason to prefer function objects is efficiency, since the compiler can *inline* the definitions of functions given as member functions of classes, like the `operator()` member given in class `multiply`, so that there is no overhead of function calling. When an ordinary function is used, one adds calling overhead to the computation the function body does.

Before leaving this topic, we should mention that in 02-11 it wasn't really necessary to define class `multiply`, since STL includes such a definition, although in a more general form:

```
template <class T>
class multiplies : public binary_function<T, T, T> {
 public:
  T operator()(const T& x, const T& y) const {
    return x * y;
  }
};
```

This class inherits from another STL component, `binary_function`, whose purpose is to hold extra information about the function, as will be discussed in Chapter 8. Using this definition, the program can be written as follows:

"ex02-12.cpp" 0 ≡

```
// Using the generic accumulate algorithm to compute a product,
// using a function object.
#include <iostream>
#include <vector>
#include <numeric>
#include <functional>
#include <assert.h>
using namespace std;

int main()
{
  cout << "Using generic accumulate algorithm to "
       << "compute a product." << endl;

  int x[5] = {2, 3, 5, 7, 11};

  // Initialize vector1 to x[0] through x[4]:
  vector<int> vector1(x, x+5);

  int product = accumulate(vector1.begin(), vector1.end(),
                           1, multiplies<int>());

  assert(product == 2310);
  return 0;
}
```

The class `multiplies` is defined in the header `function.h`, which is already included by `algo.h`, so we do not need any additional header files. With `multiplies<int>()`, we just call the default constructor of class `multiplies` instantiated with type `int`. A few of the other STL-provided function objects are shown in the fourth column of **Figure 2-1** in Section 2.3.

2.5 Adaptors

A component that modifies the interface of another component is called an *adaptor*. Adaptors are depicted in the last column of **Figure 2-1** in Section 2.3. For example, `reverse_iterator` is a component that adapts an iterator type into a new type of iterator with all the capabilities of the original but with the direction of traversal reversed. This is useful because sometimes the standard traversal order is not what's needed in a particular computation. For example, `find` returns an iterator referring to the *first* occurrence of a value in a sequence, but we might want the *last* occurrence instead. We could use `reverse` to reverse the order of the elements in the sequence and then use `find`, but we can do it without disturbing or copying the sequence by using reverse iterators. In **Figure 2-1**, a reverse iterator component is depicted as having the "wires" for `++` and `--` crossed.

Continuing with `accumulate` as an example, it might not appear too useful to traverse a sequence in reverse order to accumulate its values, since the sum

should be the same as with a forward traversal. That's true of a sequence of integers, with + as the combining function, since on integers + obeys the laws $(x + y) + z = x + (y + z)$ (associativity) and $x + y = y + x$ (commutativity), but these properties can fail for floating-point numbers because of round-off and overflow errors (associativity can fail even with `int`s because of overflow). With floating-point numbers, round-off errors are usually smaller if the numbers are added in order of increasing size; otherwise, values that are very small relative to the running sum may have no effect at all on the sum. Suppose we have a vector of values in descending order and we want to accumulate their sum. In order to add them in ascending order, we can use `accumulate` with reverse iterators:

"ex02-13.cpp" 0 ≡

```
// Demonstrating generic accumulate algorithm with a reverse
// iterator.
#include <iostream>
#include <vector>
#include <numeric>
#include <assert.h>
using namespace std;

int main()
{
  cout << "Demonstrating generic accumulate algorithm with "
       << "a reverse iterator." << endl;

  float small = (float)1.0/(1 << 26);
  float x[5] = {1.0, 3*small, 2*small, small, small};

  // Initialize vector1 to x[0] through x[4]:
  vector<float> vector1(&x[0], &x[5]);

  cout << "Values to be added: " << endl;

  vector<float>::iterator i;
  cout.precision(10);
  for (i = vector1.begin(); i != vector1.end(); ++i)
    cout << *i << endl;
  cout << endl;

  float sum = accumulate(vector1.begin(), vector1.end(),
                         (float)0.0);

  cout << "Sum accumulated from left = " <<  sum << endl;

  float sum1 = accumulate(vector1.rbegin(), vector1.rend(),
                          (float)0.0);

  cout << "Sum accumulated from right = " << (double)sum1 << endl;
  return 0;
}
```

In computing `sum1`, we use `vector` member functions `rbegin` and `rend` to obtain iterators of type `vector<float>::reverse_iterator`, which, like `vector<float>::iterator`, is defined as part of the `vector` interface. The output of this program will vary depending on the precision used in type `float`, but the value of `small` was chosen to make a difference between `sum` and `sum1` for a precision of about 8 decimal places.[4] In this case, the output is as follows:

```
Demonstrating accumulate function with a reverse iterator.
Values to be added:
1
4.470348358e-08
2.980232239e-08
1.490116119e-08
1.490116119e-08
Sum accumulated from left = 1
Sum accumulated from right = 1.000000119
```

The sum accumulated from the right, using the smaller values first, is the more accurate one.

The type `vector<float>::reverse_iterator` is actually defined using an iterator adaptor. We could have used this adaptor directly in our program by writing

```
⟨Reverse iterator specialization 25a⟩
   start(vector1.end()), finish(vector1.begin());
float sum1 = accumulate(start, finish, (float)0.0);
```

Here `start` and `finish` are declared as variables of type

```
reverse_iterator<vector<float>::iterator, float,
                 float&, ptrdiff_t>
```

in which the first template parameter, `vector<float>::iterator`, is an iterator type and the second, `float`, is the corresponding value type. The third parameter, `float&`, is a reference type for the value type, and the last parameter, `ptrdiff_t`, is a distance type for the iterator type (a type capable of representing differences between iterator values). This `reverse_iterator` type provides `++` and `--` operators, just as `vector<float>::iterator` does, but with their meanings exchanged. As a convenience, each of the container types that STL defines provides a `reverse_iterator` type already defined in this way, along with `rbegin` and `rend` member functions that return iterators of this type.

Besides reverse iterators, STL also has another kind of iterator adaptor. *Insert iterators* are provided to allow generic algorithms to operate in an "insert mode" rather than in their ordinary overwrite mode; they are applied to containers and produce output iterators. Insert iterators are especially useful for transferring a data sequence from an input stream or container to another container without having to know in advance the length of the sequence. Iterator adaptors are described more fully in Chapter 10.

STL also defines several kinds of container adaptors and function adaptors. A *stack adaptor* transforms a sequence container into a container with the more restricted interface of a last-in, first-out stack. A *queue adaptor* transforms a sequence container into a first-in, first-out queue, and a *priority queue* adaptor produces a queue in which values are accessible in an order controlled by a comparison parameter. These container adaptors are described more fully in Chapter 9.

The function adaptors STL provides include negators, binders, and adaptors for pointers to functions. A *negator* is a kind of function adaptor used to reverse the sense of predicate function objects, which are function objects that return a `bool` value. A *binder* is used to convert binary function objects into unary function objects by binding an argument to some particular value. A pointer-to-function adaptor transforms a pointer to a function into a function object; it can be used to give compiled code more flexibility than one can obtain using standard function objects (which helps to avoid the "code bloat" that can result from using too many compiled combinations of algorithms and function objects in the same program). These function adaptors are described in more detail in Chapter 11.

2.6 Allocators

Every STL container class uses an *Allocator* class to encapsulate information about the memory model the program is using. Different memory models have different requirements for pointers, references, integer sizes, and so forth. The *Allocator* class encapsulates information about pointers, constant pointers, references, constant references, sizes of objects, difference types between pointers, allocation and deallocation functions, as well as some other functions. All operations on allocators are expected to be amortized constant time.

Since memory model information can be encapsulated in an allocator, STL containers can work with different memory models simply by providing different allocators.

We do not cover allocators in detail in this book, since the default allocator class supplied with STL implementations is sufficient for most programmers' needs. For programmers who do need to define new allocators, Chapter 22 describes the information that an allocator must provide.

For further insight on the motivation for allocators being a part of STL, see Ref. 13 (although it should be noted that the details of the way allocators are provided have changed since that paper was written).