Chapter 12 Dynamic Memory

With each revision of the C++ language, the library has also grown. Indeed, more than two-thirds of the text of the new standard is devoted to the library. Although we cannot cover every library facility in depth, there are core facilities that the library defines that every C++ programmer should be comfortable using. We cover these core facilities in this part.

We'll start by covering the basic IO library facilities in Chapter 8. Beyond using the library to read and write streams associated with the console window, the library defines types that let us read and write named files and do in-memory IO to `strings`.

Central to the library are a number of container classes and a family of generic algorithms that let us write programs that are succinct and efficient. The library worries about bookkeeping details—in particular, taking care of memory management—so that our programs can worry about the actual problems we need to solve.

In Chapter 3 we introduced the `vector` container type. We'll learn more about `vector` in Chapter 9, which will cover the other sequential container types as well. We'll also cover more operations provided by the `string` type. We can think of a `string` as a special kind of container that contains only characters. The `string` type supports many, but not all, of the container operations.

Chapter 10 introduces the generic algorithms. The algorithms typically operate on a range of elements in a sequential container or other sequence. The algorithms library offers efficient implementations of various classical algorithms, such as sorting and searching, and other common tasks as well. For example, there is a `copy` algorithm, which copies elements from one sequence to another; `find`, which looks for a given element; and so on. The algorithms are generic in two ways: They can be applied to different kinds of sequences, and those sequences may contain elements of most types.

The library also provides several associative containers, which are the topic of Chapter 11. Elements in an associative container are accessed by key. The associative containers share many operations with the sequential containers and also define operations that are specific to the associative containers.

This part concludes with Chapter 12, which looks at language and library facilities for managing dynamic memory. This chapter covers one of the most important new library classes, which are standardized versions of smart pointers. By using smart pointers, we can make code that uses dynamic memory much more robust. This chapter closes with an extended example that uses library facilities introduced throughout Part II.

Chapter 8. The IO Library

Contents

The C++ language does not deal directly with input and output. Instead, IO is handled by a family of types defined in the standard library. These types support IO to and from devices such as files and console windows. Additional types allow in-memory IO to and from `strings`.

The IO library defines operations to read and write values of the built-in types. In addition, classes, such as `string`, typically define similar IO operations to work on objects of their class type as well.

This chapter introduces the fundamentals of the IO library. Later chapters will cover additional capabilities: Chapter 14 will look at how we can write our own input and output operators, and Chapter 17 will cover how to control formatting and how to perform random access on files.

Our programs have already used many IO library facilities. Indeed, we introduced most of these facilities in § 1.2 (p. 5):

- `istream` (input stream) type, which provides input operations
- `ostream` (output stream) type, which provides output operations
- `cin`, an `istream` object that reads the standard input
- `cout`, an `ostream` object that writes to the standard output
- `cerr`, an `ostream` object, typically used for program error messages, that writes to the standard error
- The >> operator, which is used to read input from an `istream` object
- The << operator, which is used to write output to an `ostream` object
- The `getline` function (§ 3.2.2, p. 87), which reads a line of input from a given `istream` into a given `string`

8.1. The IO Classes

The IO types and objects that we've used so far manipulate `char` data. By default these objects are connected to the user's console window. Of course, real programs cannot be limited to doing IO solely to or from a console window. Programs often need to read or write named files. Moreover, it can be convenient to use IO

operations to process the characters in a `string`. Applications also may have to read and write languages that require wide-character support.

To support these different kinds of IO processing, the library defines a collection of IO types in addition to the `istream` and `ostream` types that we have already used. These types, which are listed in Table 8.1, are defined in three separate headers: `iostream` defines the basic types used to read from and write to a stream, `fstream` defines the types used to read and write named files, and `sstream` defines the types used to read and write in-memory `strings`.

Table 8.1. IO Library Types and Headers

Header	Type
`iostream`	`istream`, `wistream` reads from a stream
	`ostream`, `wostream` writes to a stream
	`iostream`, `wiostream` reads and writes a stream
`fstream`	`ifstream`, `wifstream` reads from a file
	`ofstream`, `wofstream` writes to a file
	`fstream`, `wfstream` reads and writes a file
`sstream`	`istringstream`, `wistringstream` reads from a `string`
	`ostringstream`, `wostringstream` writes to a `string`
	`stringstream`, `wstringstream` reads and writes a `string`

To support languages that use wide characters, the library defines a set of types and objects that manipulate `wchar_t` data (§ 2.1.1, p. 32). The names of the wide-character versions begin with a `w`. For example, `wcin`, `wcout`, and `wcerr` are the wide-character objects that correspond to `cin`, `cout`, and `cerr`, respectively. The wide-character types and objects are defined in the same header as the plain `char` types. For example, the `fstream` header defines both the `ifstream` and `wifstream` types.

Relationships among the IO Types

Conceptually, neither the kind of device nor the character size affects the IO operations we want to perform. For example, we'd like to use >> to read data regardless of whether we're reading a console window, a disk file, or a `string`. Similarly, we'd like to use that operator regardless of whether the characters we read fit in a `char` or require a `wchar_t`.

The library lets us ignore the differences among these different kinds of streams by using **inheritance**. As with templates (§ 3.3, p. 96), we can use classes related by inheritance without understanding the details of how inheritance works. We'll cover how C++ supports inheritance in Chapter 15 and in § 18.3 (p. 802).

Briefly, inheritance lets us say that a particular class inherits from another class. Ordinarily, we can use an object of an inherited class as if it were an object of the same type as the class from which it inherits.

The types `ifstream` and `istringstream` inherit from `istream`. Thus, we can use objects of type `ifstream` or `istringstream` as if they were `istream` objects. We can use objects of these types in the same ways as we have used `cin`. For example, we can call `getline` on an `ifstream` or `istringstream` object, and we can use the >> to read data from an `ifstream` or `istringstream`. Similarly, the types `ofstream` and `ostringstream` inherit from `ostream`. Therefore, we can use objects of these types in the same ways that we have used `cout`.

Note

Everything that we cover in the remainder of this section applies equally to plain streams, file streams, and `string` streams and to the `char` or wide-character stream versions.

8.1.1. No Copy or Assign for IO Objects

As we saw in § 7.1.3 (p. 261), we cannot copy or assign objects of the IO types:

Click here to view code image

```
ofstream out1, out2;
out1 = out2;                  // error: cannot assign stream objects
ofstream print(ofstream);     // error: can't initialize the ofstream parameter
out2 = print(out2);           // error: cannot copy stream objects
```

Because we can't copy the IO types, we cannot have a parameter or return type that is one of the stream types (§ 6.2.1, p. 209). Functions that do IO typically pass and return the stream through references. Reading or writing an IO object changes its state, so the reference must not be `const`.

8.1.2. Condition States

Inherent in doing IO is the fact that errors can occur. Some errors are recoverable; others occur deep within the system and are beyond the scope of a program to correct. The IO classes define functions and flags, listed in Table 8.2, that let us access and manipulate the **condition state** of a stream.

Table 8.2. IO Library Condition State

strm::iostate	*strm* is one of the IO types listed in Table 8.1 (p. 310). iostate is a machine-dependent integral type that represents the condition state of a stream.
strm::badbit	*strm*::iostate value used to indicate that a stream is corrupted.
strm::failbit	*strm*::iostate value used to indicate that an IO operation failed.
strm::eofbit	*strm*::iostate value used to indicate that a stream hit end-of-file.
strm::goodbit	*strm*::iostate value used to indicate that a stream is not in an error state. This value is guaranteed to be zero.
s.eof()	true if eofbit in the stream s is set.
s.fail()	true if failbit or badbit in the stream s is set.
s.bad()	true if badbit in the stream s is set.
s.good()	true if the stream s is in a valid state.
s.clear()	Reset all condition values in the stream s to valid state. Returns void.
s.clear(flags)	Reset the condition of s to flags. Type of flags is *strm*::iostate. Returns void.
s.setstate(flags)	Adds specified condition(s) to s. Type of flags is *strm*::iostate. Returns void.
s.rdstate()	Returns current condition of s as a *strm*::iostate value.

As an example of an IO error, consider the following code:

```
int ival;
cin >> ival;
```

If we enter Boo on the standard input, the read will fail. The input operator expected to read an int but got the character B instead. As a result, cin will be put in an error state. Similarly, cin will be in an error state if we enter an end-of-file.

Once an error has occurred, subsequent IO operations on that stream will fail. We can read from or write to a stream only when it is in a non-error state. Because a stream might be in an error state, code ordinarily should check whether a stream is okay before attempting to use it. The easiest way to determine the state of a stream object is to use that object as a condition:

```
while (cin >> word)
    // ok: read operation successful . . .
```

The while condition checks the state of the stream returned from the >> expression. If that input operation succeeds, the state remains valid and the condition will succeed.

Interrogating the State of a Stream

Using a stream as a condition tells us only whether the stream is valid. It does not tell us what happened. Sometimes we also need to know why the stream is invalid. For example, what we do after hitting end-of-file is likely to differ from what we'd do if

we encounter an error on the IO device.

The IO library defines a machine-dependent integral type named `iostate` that it uses to convey information about the state of a stream. This type is used as a collection of bits, in the same way that we used the `quiz1` variable in § 4.8 (p. 154). The IO classes define four `constexpr` values (§ 2.4.4, p. 65) of type `iostate` that represent particular bit patterns. These values are used to indicate particular kinds of IO conditions. They can be used with the bitwise operators (§ 4.8, p. 152) to test or set multiple flags in one operation.

The `badbit` indicates a system-level failure, such as an unrecoverable read or write error. It is usually not possible to use a stream once `badbit` has been set. The `failbit` is set after a recoverable error, such as reading a character when numeric data was expected. It is often possible to correct such problems and continue using the stream. Reaching end-of-file sets both `eofbit` and `failbit`. The `goodbit`, which is guaranteed to have the value 0, indicates no failures on the stream. If any of `badbit`, `failbit`, or `eofbit` are set, then a condition that evaluates that stream will fail.

The library also defines a set of functions to interrogate the state of these flags. The `good` operation returns `true` if none of the error bits is set. The `bad`, `fail`, and `eof` operations return `true` when the corresponding bit is on. In addition, `fail` returns `true` if `bad` is set. By implication, the right way to determine the overall state of a stream is to use either `good` or `fail`. Indeed, the code that is executed when we use a stream as a condition is equivalent to calling `!fail()`. The `eof` and `bad` operations reveal only whether those specific errors have occurred.

Managing the Condition State

The `rdstate` member returns an `iostate` value that corresponds to the current state of the stream. The `setstate` operation turns on the given condition bit(s) to indicate that a problem occurred. The `clear` member is overloaded (§ 6.4, p. 230): One version takes no arguments and a second version takes a single argument of type `iostate`.

The version of `clear` that takes no arguments turns off all the failure bits. After `clear()`, a call to `good` returns `true`. We might use these members as follows:

Click here to view code image

```
// remember the current state of cin
auto old_state = cin.rdstate();  // remember the current state of cin
cin.clear();                     // make cin valid
process_input(cin);              // use cin
cin.setstate(old_state);         // now reset cin to its old state
```

The version of `clear` that takes an argument expects an `iostate` value that represents the new state of the stream. To turn off a single condition, we use the

`rdstate` member and the bitwise operators to produce the desired new state.

For example, the following turns off `failbit` and `badbit` but leaves `eofbit` untouched:

Click here to view code image

```
// turns off failbit and badbit but all other bits unchanged
cin.clear(cin.rdstate() & ~cin.failbit & ~cin.badbit);
```

Exercises Section 8.1.2

Exercise 8.1: Write a function that takes and returns an `istream&`. The function should read the stream until it hits end-of-file. The function should print what it reads to the standard output. Reset the stream so that it is valid before returning the stream.

Exercise 8.2: Test your function by calling it, passing `cin` as an argument.

Exercise 8.3: What causes the following `while` to terminate?

```
while (cin >> i) /*  ...    */
```

8.1.3. Managing the Output Buffer

Each output stream manages a buffer, which it uses to hold the data that the program reads and writes. For example, when the following code is executed

```
os << "please enter a value: ";
```

the literal string might be printed immediately, or the operating system might store the data in a buffer to be printed later. Using a buffer allows the operating system to combine several output operations from our program into a single system-level write. Because writing to a device can be time-consuming, letting the operating system combine several output operations into a single write can provide an important performance boost.

There are several conditions that cause the buffer to be flushed—that is, to be written—to the actual output device or file:

- The program completes normally. All output buffers are flushed as part of the `return` from `main`.
- At some indeterminate time, the buffer can become full, in which case it will be flushed before writing the next value.
- We can flush the buffer explicitly using a manipulator such as `endl` (§ 1.2, p. 7).
- We can use the `unitbuf` manipulator to set the stream's internal state to

empty the buffer after each output operation. By default, `unitbuf` is set for `cerr`, so that writes to `cerr` are flushed immediately.

- An output stream might be tied to another stream. In this case, the buffer of the tied stream is flushed whenever the tied stream is read or written. By default, `cin` and `cerr` are both tied to `cout`. Hence, reading `cin` or writing to `cerr` flushes the buffer in `cout`.

Flushing the Output Buffer

Our programs have already used the `endl` manipulator, which ends the current line and flushes the buffer. There are two other similar manipulators: `flush` and `ends`. `flush` flushes the stream but adds no characters to the output; `ends` inserts a null character into the buffer and then flushes it:

Click here to view code image

```
cout << "hi!" << endl;    // writes hi and a newline, then flushes the buffer
cout << "hi!" << flush;   // writes hi, then flushes the buffer; adds no data
cout << "hi!" << ends;    // writes hi and a null, then flushes the buffer
```

The unitbuf Manipulator

If we want to flush after every output, we can use the `unitbuf` manipulator. This manipulator tells the stream to do a `flush` after every subsequent write. The `nounitbuf` manipulator restores the stream to use normal, system-managed buffer flushing:

Click here to view code image

```
cout << unitbuf;        // all writes will be flushed immediately
// any output is flushed immediately, no buffering
cout << nounitbuf;      // returns to normal buffering
```

> **Caution: Buffers Are Not Flushed If the Program Crashes**
>
> Output buffers are *not* flushed if the program terminates abnormally. When a program crashes, it is likely that data the program wrote may be sitting in an output buffer waiting to be printed.
>
> When you debug a program that has crashed, it is essential to make sure that any output you *think* should have been written was actually flushed. Countless hours of programmer time have been wasted tracking through code that appeared not to have executed when in fact the buffer had not been flushed and the output was pending when the program crashed.

Tying Input and Output Streams Together

When an input stream is tied to an output stream, any attempt to read the input stream will first flush the buffer associated with the output stream. The library ties `cout` to `cin`, so the statement

```
cin >> ival;
```

causes the buffer associated with `cout` to be flushed.

Note

Interactive systems usually should tie their input stream to their output stream. Doing so means that all output, which might include prompts to the user, will be written before attempting to read the input.

There are two overloaded (§ 6.4, p. 230) versions of `tie`: One version takes no argument and returns a pointer to the output stream, if any, to which this object is currently tied. The function returns the null pointer if the stream is not tied.

The second version of `tie` takes a pointer to an `ostream` and ties itself to that `ostream`. That is, `x.tie(&o)` ties the stream `x` to the output stream `o`.

We can tie either an `istream` or an `ostream` object to another `ostream`:

Click here to view code image

```
cin.tie(&cout);      // illustration only: the library ties cin and cout for us
// old_tie points to the stream (if any) currently tied to cin
ostream *old_tie = cin.tie(nullptr); // cin is no longer tied
// ties cin and cerr; not a good idea because cin should be tied to cout
cin.tie(&cerr);      // reading cin flushes cerr, not cout
cin.tie(old_tie); // reestablish normal tie between cin and cout
```

To tie a given stream to a new output stream, we pass `tie` a pointer to the new stream. To untie the stream completely, we pass a null pointer. Each stream can be tied to at most one stream at a time. However, multiple streams can tie themselves to the same `ostream`.

8.2. File Input and Output

The `fstream` header defines three types to support file IO: **ifstream** to read from a given file, **ofstream** to write to a given file, and **fstream**, which reads and writes a given file. In § 17.5.3 (p. 763) we'll describe how to use the same file for both input

and output.

These types provide the same operations as those we have previously used on the objects `cin` and `cout`. In particular, we can use the IO operators (`<<` and `>>`) to read and write files, we can use `getline` (§ 3.2.2, p. 87) to read an `ifstream`, and the material covered in § 8.1 (p. 310) applies to these types.

In addition to the behavior that they inherit from the `iostream` types, the types defined in `fstream` add members to manage the file associated with the stream. These operations, listed in Table 8.3, can be called on objects of `fstream`, `ifstream`, or `ofstream` but not on the other IO types.

Table 8.3. fstream-Specific Operations

fstream `fstrm;`	Creates an unbound file stream. *fstream* is one of the types defined in the `fstream` header.
fstream `fstrm(s);`	Creates an *fstream* and opens the file named `s`. `s` can have type `string` or can be a pointer to a C-style character string (§ 3.5.4, p. 122). These constructors are `explicit` (§ 7.5.4, p. 296). The default file `mode` depends on the type of *fstream*.
fstream `fstrm(s, mode);`	Like the previous constructor, but opens `s` in the given `mode`.
`fstrm.open(s)` `fstrm.open(s, mode)`	Opens the file named by the `s` and binds that file to `fstrm`. `s` can be a `string` or a pointer to a C-style character string. The default file `mode` depends on the type of *fstream*. Returns `void`.
`fstrm.close()`	Closes the file to which `fstrm` is bound. Returns `void`.
`fstrm.is_open()`	Returns a `bool` indicating whether the file associated with `fstrm` was successfully opened and has not been closed.

8.2.1. Using File Stream Objects

When we want to read or write a file, we define a file stream object and associate that object with the file. Each file stream class defines a member function named `open` that does whatever system-specific operations are required to locate the given file and open it for reading or writing as appropriate.

When we create a file stream, we can (optionally) provide a file name. When we supply a file name, `open` is called automatically:

Click here to view code image

```
ifstream in(ifile); // construct an ifstream and open the given file
ofstream out;       // output file stream that is not associated with any file
```

This code defines `in` as an input stream that is initialized to read from the file named by the `string` argument `ifile`. It defines `out` as an output stream that is not yet associated with a file. With the new standard, file names can be either library

strings or C-style character arrays (§ 3.5.4, p. 122). Previous versions of the library allowed only C-style character arrays.

Using an fstream in Place of an iostream&

As we noted in § 8.1 (p. 311), we can use an object of an inherited type in places where an object of the original type is expected. This fact means that functions that are written to take a reference (or pointer) to one of the iostream types can be called on behalf of the corresponding fstream (or sstream) type. That is, if we have a function that takes an ostream&, we can call that function passing it an ofstream object, and similarly for istream& and ifstream.

For example, we can use the read and print functions from § 7.1.3 (p. 261) to read from and write to named files. In this example, we'll assume that the names of the input and output files are passed as arguments to main (§ 6.2.5, p. 218):

Click here to view code image

```
ifstream input(argv[1]);    // open the file of sales transactions
ofstream output(argv[2]);   // open the output file
Sales_data total;           // variable to hold the running sum
if (read(input, total)) {   // read the first transaction
    Sales_data trans;       // variable to hold data for the next transaction
    while(read(input, trans)) {     // read the remaining transactions
        if (total.isbn() == trans.isbn()) //   check isbns
            total.combine(trans);   // update the running total
        else {
            print(output, total) << endl; //   print the results
            total = trans;          // process the next book
        }
    }
    print(output, total) << endl;   // print the last transaction
} else                              // there was no input
    cerr << "No data?!" << endl;
```

Aside from using named files, this code is nearly identical to the version of the addition program on page 255. The important part is the calls to read and to print. We can pass our fstream objects to these functions even though the parameters to those functions are defined as istream& and ostream&, respectively.

The open and close Members

When we define an empty file stream object, we can subsequently associate that object with a file by calling open:

Click here to view code image

```
ifstream in(ifile); // construct an ifstreamand open the given file
ofstream out;       // output file stream that is not associated with any file
out.open(ifile + ".copy");  // open the specified file
```

If a call to open fails, `failbit` is set (§ 8.1.2, p. 312). Because a call to `open` might fail, it is usually a good idea to verify that the `open` succeeded:

Click here to view code image

```
if (out)      // check that the open succeeded
    // the open succeeded, so we can use the file
```

This condition is similar to those we've used on `cin`. If the `open` fails, this condition will fail and we will not attempt to use `in`.

Once a file stream has been opened, it remains associated with the specified file. Indeed, calling `open` on a file stream that is already open will fail and set `failbit`. Subsequent attempts to use that file stream will fail. To associate a file stream with a different file, we must first close the existing file. Once the file is closed, we can open a new one:

Click here to view code image

```
in.close();                 // close the file
in.open(ifile + "2");       // open another file
```

If the `open` succeeds, then `open` sets the stream's state so that `good()` is `true`.

Automatic Construction and Destruction

Consider a program whose `main` function takes a list of files it should process (§ 6.2.5, p. 218). Such a program might have a loop like the following:

Click here to view code image

```
// for each file passed to the program
for (auto p = argv + 1; p != argv + argc; ++p) {
    ifstream input(*p);    // create input and open the file
    if (input) {           // if the file is ok, "process" this file
        process(input);
    } else
        cerr << "couldn't open: " + string(*p);
} // input goes out of scope and is destroyed on each iteration
```

Each iteration constructs a new `ifstream` object named `input` and opens it to read the given file. As usual, we check that the `open` succeeded. If so, we pass that file to a function that will read and process the input. If not, we print an error message and continue.

Because `input` is local to the `while`, it is created and destroyed on each iteration (§ 5.4.1, p. 183). When an `fstream` object goes out of scope, the file it is bound to is automatically closed. On the next iteration, `input` is created anew.

Note

When an `fstream` object is destroyed, `close` is called automatically.

Exercises Section 8.2.1

Exercise 8.4: Write a function to open a file for input and read its contents into a `vector` of `strings`, storing each line as a separate element in the `vector`.

Exercise 8.5: Rewrite the previous program to store each word in a separate element.

Exercise 8.6: Rewrite the bookstore program from § 7.1.1 (p. 256) to read its transactions from a file. Pass the name of the file as an argument to `main` (§ 6.2.5, p. 218).

8.2.2. File Modes

Each stream has an associated **file mode** that represents how the file may be used. Table 8.4 lists the file modes and their meanings.

Table 8.4. File Modes

`in`	Open for input
`out`	Open for output
`app`	Seek to the end before every write
`ate`	Seek to the end immediately after the open
`trunc`	Truncate the file
`binary`	Do IO operations in binary mode

We can supply a file mode whenever we open a file—either when we call `open` or when we indirectly open the file when we initialize a stream from a file name. The modes that we can specify have the following restrictions:

- `out` may be set only for an `ofstream` or `fstream` object.
- `in` may be set only for an `ifstream` or `fstream` object.

- `trunc` may be set only when `out` is also specified.
- `app` mode may be specified so long as `trunc` is not. If `app` is specified, the file is always opened in output mode, even if `out` was not explicitly specified.
- By default, a file opened in `out` mode is truncated even if we do not specify `trunc`. To preserve the contents of a file opened with `out`, either we must also specify `app`, in which case we can write only at the end of the file, or we must also specify `in`, in which case the file is open for both input and output (§ 17.5.3 (p. 763) will cover using the same file for input and output).
- The `ate` and `binary` modes may be specified on any file stream object type and in combination with any other file modes.

Each file stream type defines a default file mode that is used whenever we do not otherwise specify a mode. Files associated with an `ifstream` are opened in `in` mode; files associated with an `ofstream` are opened in `out` mode; and files associated with an `fstream` are opened with both `in` and `out` modes.

Opening a File in out Mode Discards Existing Data

By default, when we open an `ofstream`, the contents of the file are discarded. The only way to prevent an `ostream` from emptying the given file is to specify `app`:

Click here to view code image

```
// file1 is truncated in each of these cases
ofstream out("file1");    // out and trunc are implicit
ofstream out2("file1", ofstream::out);    // trunc is implicit
ofstream out3("file1", ofstream::out | ofstream::trunc);

// to preserve the file's contents, we must explicitly specify app mode
ofstream app("file2", ofstream::app);    // out is implicit
ofstream app2("file2", ofstream::out | ofstream::app);
```

Warning

The only way to preserve the existing data in a file opened by an `ofstream` is to specify `app` or `in` mode explicitly.

File Mode Is Determined Each Time open Is Called

The file mode of a given stream may change each time a file is opened.

Click here to view code image

```
ofstream out;     // no file mode is set
out.open("scratchpad"); // mode implicitly out and trunc
out.close();      // close out so we can use it for a different file
out.open("precious", ofstream::app);   // mode is out and app
out.close();
```

The first call to `open` does not specify an output mode explicitly; this file is implicitly opened in `out` mode. As usual, `out` implies `trunc`. Therefore, the file named `scratchpad` in the current directory will be truncated. When we open the file named `precious`, we ask for append mode. Any data in the file remains, and all writes are done at the end of the file.

Note

Any time `open` is called, the file mode is set, either explicitly or implicitly. Whenever a mode is not specified, the default value is used.

Exercises Section 8.2.2

Exercise 8.7: Revise the bookstore program from the previous section to write its output to a file. Pass the name of that file as a second argument to `main`.

Exercise 8.8: Revise the program from the previous exercise to append its output to its given file. Run the program on the same output file at least twice to ensure that the data are preserved.

8.3. `string` Streams

The `sstream` header defines three types to support in-memory IO; these types read from or write to a `string` as if the `string` were an IO stream.

The **`istringstream`** type reads a `string`, **`ostringstream`** writes a `string`, and **`stringstream`** reads and writes the `string`. Like the `fstream` types, the types defined in `sstream` inherit from the types we have used from the `iostream` header. In addition to the operations they inherit, the types defined in `sstream` add members to manage the `string` associated with the stream. These operations are listed in Table 8.5. They may be called on `stringstream` objects but not on the other IO types.

Table 8.5. `stringstream`-Specific Operations

sstream `strm;`	`strm` is an unbound `stringstream`. *sstream* is one of the types defined in the `sstream` header.
sstream `strm(s);`	`strm` is an *sstream* that holds a copy of the `string` `s`. This constructor is `explicit` (§ 7.5.4, p. 296).
`strm.str()`	Returns a copy of the `string` that `strm` holds.
`strm.str(s)`	Copies the `string` `s` into `strm`. Returns `void`.

Note that although `fstream` and `sstream` share the interface to `iostream`, they have no other interrelationship. In particular, we cannot use `open` and `close` on a `stringstream`, nor can we use `str` on an `fstream`.

8.3.1. Using an istringstream

An `istringstream` is often used when we have some work to do on an entire line, and other work to do with individual words within a line.

As one example, assume we have a file that lists people and their associated phone numbers. Some people have only one number, but others have several—a home phone, work phone, cell number, and so on. Our input file might look like the following:

Click here to view code image

```
morgan 2015552368 8625550123
drew 9735550130
lee 6095550132 2015550175 8005550000
```

Each record in this file starts with a name, which is followed by one or more phone numbers. We'll start by defining a simple class to represent our input data:

Click here to view code image

```
// members are public by default; see § 7.2 (p. 268)
struct PersonInfo {
    string name;
    vector<string> phones;
};
```

Objects of type `PersonInfo` will have one member that represents the person's name and a `vector` holding a varying number of associated phone numbers.

Our program will read the data file and build up a `vector` of `PersonInfo`. Each element in the `vector` will correspond to one record in the file. We'll process the input in a loop that reads a record and then extracts the name and phone numbers for each person:

Click here to view code image

```
string line, word;  // will hold a line and word from input, respectively
```

```
vector<PersonInfo> people; // will hold all the records from the input
// read the input a line at a time until cin hits end-of-file (or another error)
while (getline(cin, line)) {
    PersonInfo info;        // create an object to hold this record's data
    istringstream record(line); // bind record to the line we just read
    record >> info.name;   // read the name
    while (record >> word)          // read the phone numbers
        info.phones.push_back(word);  // and store them
    people.push_back(info); // append this record to people
}
```

Here we use `getline` to read an entire record from the standard input. If the call to `getline` succeeds, then `line` holds a record from the input file. Inside the `while` we define a local `PersonInfo` object to hold data from the current record.

Next we bind an `istringstream` to the line that we just read. We can now use the input operator on that `istringstream` to read each element in the current record. We first read the name followed by a `while` loop that will read the phone numbers for that person.

The inner `while` ends when we've read all the data in `line`. This loop works analogously to others we've written to read `cin`. The difference is that this loop reads data from a `string` rather than from the standard input. When the `string` has been completely read, "end-of-file" is signaled and the next input operation on `record` will fail.

We end the outer `while` loop by appending the `PersonInfo` we just processed to the `vector`. The outer `while` continues until we hit end-of-file on `cin`.

Exercises Section 8.3.1

Exercise 8.9: Use the function you wrote for the first exercise in § 8.1.2 (p. 314) to print the contents of an `istringstream` object.

Exercise 8.10: Write a program to store each line from a file in a `vector<string>`. Now use an `istringstream` to read each element from the `vector` a word at a time.

Exercise 8.11: The program in this section defined its `istringstream` object inside the outer `while` loop. What changes would you need to make if `record` were defined outside that loop? Rewrite the program, moving the definition of `record` outside the `while`, and see whether you thought of all the changes that are needed.

Exercise 8.12: Why didn't we use in-class initializers in `PersonInfo`?

8.3.2. Using ostringstreams

An `ostringstream` is useful when we need to build up our output a little at a time but do not want to print the output until later. For example, we might want to validate and reformat the phone numbers we read in the previous example. If all the numbers are valid, we want to print a new file containing the reformatted numbers. If a person has any invalid numbers, we won't put them in the new file. Instead, we'll write an error message containing the person's name and a list of their invalid numbers.

Because we don't want to include any data for a person with an invalid number, we can't produce the output until we've seen and validated all their numbers. We can, however, "write" the output to an in-memory `ostringstream`:

Click here to view code image

```
for (const auto &entry : people) {     // for each entry in people
    ostringstream formatted, badNums; // objects created on each loop
    for (const auto &nums : entry.phones) { // for each number
        if (!valid(nums)) {
            badNums << " " << nums;  // string in badNums
        } else
            // "writes" to formatted's string
            formatted << " " << format(nums);
    }
    if (badNums.str().empty())       // there were no bad numbers
        os << entry.name << " "      // print the name
           << formatted.str() << endl; // and reformatted numbers
    else                             // otherwise, print the name and bad numbers
        cerr << "input error: " << entry.name
             << " invalid number(s) " << badNums.str() <<
endl;
}
```

In this program, we've assumed two functions, `valid` and `format`, that validate and reformat phone numbers, respectively. The interesting part of the program is the use of the string streams `formatted` and `badNums`. We use the normal output operator (<<) to write to these objects. But, these "writes" are really `string` manipulations. They add characters to the `strings` inside `formatted` and `badNums`, respectively.

Exercises Section 8.3.2

Exercise 8.13: Rewrite the phone number program from this section to read from a named file rather than from `cin`.

Exercise 8.14: Why did we declare `entry` and `nums` as `const auto &`?

Chapter Summary

C++ uses library classes to handle stream-oriented input and output:

- The `iostream` classes handle IO to console
- The `fstream` classes handle IO to named files
- The `stringstream` classes do IO to in-memory `strings`

The `fstream` and `stringstream` classes are related by inheritance to the `iostream` classes. The input classes inherit from `istream` and the output classes from `ostream`. Thus, operations that can be performed on an `istream` object can also be performed on either an `ifstream` or an `istringstream`. Similarly for the output classes, which inherit from `ostream`.

Each IO object maintains a set of condition states that indicate whether IO can be done through this object. If an error is encountered—such as hitting end-of-file on an input stream—then the object's state will be such that no further input can be done until the error is rectified. The library provides a set of functions to set and test these states.

Defined Terms

condition state Flags and associated functions usable by any of the stream classes that indicate whether a given stream is usable.

file mode Flags defined by the `fstream` classes that are specified when opening a file and control how a file can be used.

file stream Stream object that reads or writes a named file. In addition to the normal `iostream` operations, file streams also define `open` and `close` members. The `open` member takes a `string` or a C-style character string that names the file to open and an optional open mode argument. The `close` member closes the file to which the stream is attached. It must be called before another file can be `opened`.

fstream File stream that reads and writes to the same file. By default `fstreams` are opened with `in` and `out` mode set.

ifstream File stream that reads an input file. By default `ifstreams` are opened with `in` mode set.

inheritance Programming feature that lets a type inherit the interface of another type. The `ifstream` and `istringstream` classes inherit from `istream` and the `ofstream` and `ostringstream` classes inherit from `ostream`. Chapter 15 covers inheritance.

istringstream String stream that reads a given `string`.

ofstream File stream that writes to an output file. By default, `ofstreams` are

opened with `out` mode set.

ostringstream String stream that writes to a given `string`.

string stream Stream object that reads or writes a `string`. In addition to the normal `iostream` operations, string streams define an overloaded member named `str`. Calling `str` with no arguments returns the `string` to which the string stream is attached. Calling it with a `string` attaches the string stream to a copy of that `string`.

stringstream String stream that reads and writes to a given `string`.

Chapter 9. Sequential Containers

Contents

This chapter expands on the material from Chapter 3 and completes our discussion of the standard-library sequential containers. The order of the elements in a sequential container corresponds to the positions in which the elements are added to the container. The library also defines several associative containers, which hold elements whose position depends on a key associated with each element. We'll cover operations specific to the associative containers in Chapter 11.

The container classes share a common interface, which each of the containers extends in its own way. This common interface makes the library easier to learn; what we learn about one kind of container applies to another. Each kind of container offers a different set of performance and functionality trade-offs.

A container holds a collection of objects of a specified type. The **sequential containers** let the programmer control the order in which the elements are stored and accessed. That order does not depend on the values of the elements. Instead, the order corresponds to the position at which elements are put into the container. By contrast, the ordered and unordered associative containers, which we cover in Chapter 11, store their elements based on the value of a key.

The library also provides three container adaptors, each of which adapts a container type by defining a different interface to the container's operations. We cover the adaptors at the end of this chapter.

Note

This chapter builds on the material covered in § 3.2, § 3.3, and § 3.4. We assume that the reader is familiar with the material covered there.

9.1. Overview of the Sequential Containers

The sequential containers, which are listed in Table 9.1, all provide fast sequential access to their elements. However, these containers offer different performance trade-offs relative to

- The costs to add or delete elements to the container
- The costs to perform nonsequential access to elements of the container

Table 9.1. Sequential Container Types

`vector`	Flexible-size array. Supports fast random access. Inserting or deleting elements other than at the back may be slow.
`deque`	Double-ended queue. Supports fast random access. Fast insert/delete at front or back.
`list`	Doubly linked list. Supports only bidirectional sequential access. Fast insert/delete at any point in the `list`.
`forward_list`	Singly linked list. Supports only sequential access in one direction. Fast insert/delete at any point in the list.
`array`	Fixed-size array. Supports fast random access. Cannot add or remove elements.
`string`	A specialized container, similar to `vector`, that contains characters. Fast random access. Fast insert/delete at the back.

With the exception of `array`, which is a fixed-size container, the containers provide efficient, flexible memory management. We can add and remove elements, growing and shrinking the size of the container. The strategies that the containers use for storing their elements have inherent, and sometimes significant, impact on the efficiency of these operations. In some cases, these strategies also affect whether a particular container supplies a particular operation.

For example, `string` and `vector` hold their elements in contiguous memory. Because elements are contiguous, it is fast to compute the address of an element from its index. However, adding or removing elements in the middle of one of these

containers takes time: All the elements after the one inserted or removed have to be moved to maintain contiguity. Moreover, adding an element can sometimes require that additional storage be allocated. In that case, every element must be moved into the new storage.

The `list` and `forward_list` containers are designed to make it fast to add or remove an element anywhere in the container. In exchange, these types do not support random access to elements: We can access an element only by iterating through the container. Moreover, the memory overhead for these containers is often substantial, when compared to `vector`, `deque`, and `array`.

A `deque` is a more complicated data structure. Like `string` and `vector`, `deque` supports fast random access. As with `string` and `vector`, adding or removing elements in the middle of a `deque` is a (potentially) expensive operation. However, adding or removing elements at either end of the `deque` is a fast operation, comparable to adding an element to a `list` or `forward_list`.

The `forward_list` and `array` types were added by the new standard. An `array` is a safer, easier-to-use alternative to built-in arrays. Like built-in arrays, library `array`s have fixed size. As a result, `array` does not support operations to add and remove elements or to resize the container. A `forward_list` is intended to be comparable to the best handwritten, singly linked list. Consequently, `forward_list` does not have the `size` operation because storing or computing its size would entail overhead compared to a handwritten list. For the other containers, `size` is guaranteed to be a fast, constant-time operation.

Note

For reasons we'll explain in § 13.6 (p. 531), the new library containers are dramatically faster than in previous releases. The library containers almost certainly perform as well as (and usually better than) even the most carefully crafted alternatives. Modern C++ programs should use the library containers rather than more primitive structures like arrays.

Deciding Which Sequential Container to Use

Tip

Ordinarily, it is best to use `vector` unless there is a good reason to prefer another container.

There are a few rules of thumb that apply to selecting which container to use:

- Unless you have a reason to use another container, use a `vector`.
- If your program has lots of small elements and space overhead matters, don't use `list` or `forward_list`.
- If the program requires random access to elements, use a `vector` or a `deque`.
- If the program needs to insert or delete elements in the middle of the container, use a `list` or `forward_list`.
- If the program needs to insert or delete elements at the front and the back, but not in the middle, use a `deque`.
- If the program needs to insert elements in the middle of the container only while reading input, and subsequently needs random access to the elements:

– First, decide whether you actually need to add elements in the middle of a container. It is often easier to append to a `vector` and then call the library `sort` function (which we shall cover in § 10.2.3 (p. 384)) to reorder the container when you're done with input.

– If you must insert into the middle, consider using a `list` for the input phase. Once the input is complete, copy the `list` into a `vector`.

What if the program needs random access *and* needs to insert and delete elements in the middle of the container? This decision will depend on the relative cost of accessing the elements in a `list` or `forward_list` versus the cost of inserting or deleting elements in a `vector` or `deque`. In general, the predominant operation of the application (whether it does more access or more insertion or deletion) will determine the choice of container type. In such cases, performance testing the application using both containers will probably be necessary.

Best Practices

If you're not sure which container to use, write your code so that it uses only operations common to both `vectors` and `lists`: Use iterators, not subscripts, and avoid random access to elements. That way it will be easy to use either a `vector` or a `list` as necessary.

Exercises Section 9.1

Exercise 9.1: Which is the most appropriate—a `vector`, a `deque`, or a `list`—for the following program tasks? Explain the rationale for your choice. If there is no reason to prefer one or another container, explain why not.

(a) Read a fixed number of words, inserting them in the container alphabetically as they are entered. We'll see in the next chapter that associative containers are better suited to this problem.

(b) Read an unknown number of words. Always insert new words at the back. Remove the next value from the front.

(c) Read an unknown number of integers from a file. Sort the numbers and then print them to standard output.

9.2. Container Library Overview

The operations on the container types form a kind of hierarchy:

- Some operations (Table 9.2 (p. 330)) are provided by all container types.

Table 9.2. Container Operations

Type Aliases	
`iterator`	Type of the iterator for this container type
`const_iterator`	Iterator type that can read but not change its elements
`size_type`	Unsigned integral type big enough to hold the size of the largest possible container of this container type
`difference_type`	Signed integral type big enough to hold the distance between two iterators
`value_type`	Element type
`reference`	Element's lvalue type; synonym for `value_type&`
`const_reference`	Element's `const` lvalue type (i.e., `const value_type&`)
Construction	
`C c;`	Default constructor, empty container (`array`; see p. 336)
`C c1(c2);`	Construct `c1` as a copy of `c2`
`C c(b, e);`	Copy elements from the range denoted by iterators `b` and `e`; **(not valid for `array`)**
`C c{a,b,c...};`	List initialize `c`
Assignment and swap	
`c1 = c2`	Replace elements in `c1` with those in `c2`
`c1 = {a,b,c...}`	Replace elements in `c1` with those in the list **(not valid for `array`)**
`a.swap(b)`	Swap elements in `a` with those in `b`
`swap(a, b)`	Equivalent to `a.swap(b)`
Size	
`c.size()`	Number of elements in `c` **(not valid for `forward_list`)**
`c.max_size()`	Maximum number of elements `c` can hold
`c.empty()`	`false` if `c` has any elements, `true` otherwise

Add/Remove Elements (*not valid for* `array`)	
Note: the interface to these operations varies by container type	
`c.insert(args)`	Copy element(s) as specified by *args* into `c`
`c.emplace(inits)`	Use *inits* to construct an element in `c`
`c.erase(args)`	Remove element(s) specified by *args*
`c.clear()`	Remove all elements from `c`; returns `void`
Equality and Relational Operators	
`==, !=`	Equality valid for all container types
`<, <=, >, >=`	Relationals (**not valid for unordered associative containers**)
Obtain Iterators	
`c.begin(), c.end()`	Return iterator to the first, one past the last element in `c`
`c.cbegin(), c.cend()`	Return `const_iterator`
Additional Members of Reversible Containers (not valid for `forward_list`)	
`reverse_iterator`	Iterator that addresses elements in reverse order
`const_reverse_iterator`	Reverse iterator that cannot write the elements
`c.rbegin(), c.rend()`	Return iterator to the last, one past the first element in `c`
`c.crbegin(), c.crend()`	Return `const_reverse_iterator`

- Other operations are specific to the sequential (Table 9.3 (p. 335)), the associative (Table 11.7 (p. 438)), or the unordered (Table 11.8 (p. 445)) containers.

Table 9.3. Defining and Initializing Containers

`C c;`	Default constructor. If `C` is `array`, then the elements in `c` are default-initialized; otherwise `c` is empty.
`C c1(c2)` `C c1 = c2`	`c1` is a copy of `c2`. `c1` and `c2` must have the same type (i.e., they must be the same container type and hold the same element type; for `array` must also have the same size).
`C c{a,b,c...}` `C c = {a,b,c...}`	`c` is a copy of the elements in the initializer list. Type of elements in the list must be compatible with the element type of `C`. For `array`, the list must have same number or fewer elements than the size of the `array`, any missing elements are value-initialized (§ 3.3.1, p. 98).
`C c(b, e)`	`c` is a copy of the elements in the range denoted by iterators `b` and `e`. Type of the elements must be compatible with the element type of `C`. **(Not valid for `array`.)**
Constructors that take a size are valid for sequential containers (not including `array`) only	
`C seq(n)`	`seq` has `n` value-initialized elements; this constructor is `explicit` (§ 7.5.4, p. 296). **(Not valid for `string`.)**
`C seq(n,t)`	`seq` has `n` elements with value `t`.

- Still others are common to only a smaller subset of the containers.

In this section, we'll cover aspects common to all of the containers. The remainder of this chapter will then focus solely on sequential containers; we'll cover operations specific to the associative containers in Chapter 11.

In general, each container is defined in a header file with the same name as the type. That is, `deque` is in the `deque` header, `list` in the `list` header, and so on. The containers are class templates (§ 3.3, p. 96). As with `vectors`, we must supply additional information to generate a particular container type. For most, but not all, of the containers, the information we must supply is the element type:

Click here to view code image

```
list<Sales_data>    // list that holds Sales_data objects
deque<double>       // deque that holds doubles
```

Constraints on Types That a Container Can Hold

Almost any type can be used as the element type of a sequential container. In particular, we can define a container whose element type is itself another container. We define such containers exactly as we do any other container type: We specify the element type (which in this case is a container type) inside angle brackets:

Click here to view code image

```
vector<vector<string>> lines;       // vector of vectors
```

Here `lines` is a `vector` whose elements are `vectors` of `strings`.

Note

Older compilers may require a space between the angle brackets, for example, `vector<vector<string> >`.

Although we can store almost any type in a container, some container operations impose requirements of their own on the element type. We can define a container for a type that does not support an operation-specific requirement, but we can use an operation only if the element type meets that operation's requirements.

As an example, the sequential container constructor that takes a size argument (§ 3.3.1, p. 98) uses the element type's default constructor. Some classes do not have a default constructor. We can define a container that holds objects of such types, but we cannot construct such containers using only an element count:

Click here to view code image

```
// assume noDefault is a type without a default constructor
vector<noDefault> v1(10, init); // ok: element initializer supplied
vector<noDefault> v2(10);          // error: must supply an element
initializer
```

As we describe the container operations, we'll note the additional constraints, if any, that each container operation places on the element type.

> **Exercises Section 9.2**
>
> **Exercise 9.2:** Define a `list` that holds elements that are `deques` that hold `ints`.

9.2.1. Iterators

As with the containers, iterators have a common interface: If an iterator provides an operation, then the operation is supported in the same way for each iterator that supplies that operation. For example, all the iterators on the standard container types let us access an element from a container, and they all do so by providing the dereference operator. Similarly, the iterators for the library containers all define the increment operator to move from one element to the next.

With one exception, the container iterators support all the operations listed in Table 3.6 (p. 107). The exception is that the `forward_list` iterators do not support the decrement (`--`) operator. The iterator arithmetic operations listed in Table 3.7 (p. 111) apply only to iterators for `string`, `vector`, `deque`, and `array`. We cannot use these operations on iterators for any of the other container types.

Iterator Ranges

> 
> **Note**
>
> The concept of an iterator range is fundamental to the standard library.

An **iterator range** is denoted by a pair of iterators each of which refers to an element, or to *one past the last element,* in the same container. These two iterators, often referred to as `begin` and `end`—or (somewhat misleadingly) as `first` and `last`—mark a range of elements from the container.

The name `last`, although commonly used, is a bit misleading, because the second iterator never refers to the last element of the range. Instead, it refers to a point one past the last element. The elements in the range include the element denoted by `first` and every element from `first` up to but not including `last`.

This element range is called a **left-inclusive interval**. The standard mathematical notation for such a range is

```
[ begin, end)
```

indicating that the range begins with `begin` and ends with, but does not include, `end`. The iterators `begin` and `end` must refer to the same container. The iterator `end` may be equal to `begin` but must not refer to an element before the one denoted by `begin`.

Requirements on Iterators Forming an Iterator Range

Two iterators, `begin` and `end`, form an iterator range, if

- They refer to elements of, or one past the end of, the same container, and
- It is possible to reach `end` by repeatedly incrementing `begin`. In other words, `end` must not precede `begin`.

Warning

The compiler cannot enforce these requirements. It is up to us to ensure that our programs follow these conventions.

Programming Implications of Using Left-Inclusive Ranges

The library uses left-inclusive ranges because such ranges have three convenient properties. Assuming `begin` and `end` denote a valid iterator range, then

- If `begin` equals `end`, the range is empty
- If `begin` is not equal to `end`, there is at least one element in the range, and `begin` refers to the first element in that range
- We can increment `begin` some number of times until `begin == end`

These properties mean that we can safely write loops such as the following to process a range of elements:

Click here to view code image

```
while (begin   != end) {
    *begin =  val;     // ok: range isn't empty so  begin  denotes an element
    ++begin;           // advance the iterator to get the next element
}
```

Given that `begin` and `end` form a valid iterator range, we know that if `begin == end`, then the range is empty. In this case, we exit the loop. If the range is nonempty, we know that `begin` refers to an element in this nonempty range. Therefore, inside the body of the `while`, we know that it is safe to dereference `begin` because `begin`

must refer to an element. Finally, because the loop body increments `begin`, we also know the loop will eventually terminate.

Exercises Section 9.2.1

Exercise 9.3: What are the constraints on the iterators that form iterator ranges?

Exercise 9.4: Write a function that takes a pair of iterators to a `vector<int>` and an `int` value. Look for that value in the range and return a `bool` indicating whether it was found.

Exercise 9.5: Rewrite the previous program to return an iterator to the requested element. Note that the program must handle the case where the element is not found.

Exercise 9.6: What is wrong with the following program? How might you correct it?

Click here to view code image

```
list<int> lst1;
list<int>::iterator iter1 = lst1.begin(),
                    iter2 = lst1.end();
while (iter1 < iter2) /* ... */
```

9.2.2. Container Type Members

Each container defines several types, shown in Table 9.2 (p. 330). We have already used three of these container-defined types: `size_type` (§ 3.2.2, p. 88), `iterator`, and `const_iterator` (§ 3.4.1, p. 108).

In addition to the iterator types we've already used, most containers provide reverse iterators. Briefly, a reverse iterator is an iterator that goes backward through a container and inverts the meaning of the iterator operations. For example, saying ++ on a reverse iterator yields the previous element. We'll have more to say about reverse iterators in § 10.4.3 (p. 407).

The remaining type aliases let us use the type of the elements stored in a container without knowing what that type is. If we need the element type, we refer to the container's `value_type`. If we need a reference to that type, we use `reference` or `const_reference`. These element-related type aliases are most useful in generic programs, which we'll cover in Chapter 16.

To use one of these types, we must name the class of which they are a member:

Click here to view code image

```
// iter is the iterator type defined by list<string>
list<string>::iterator iter;
```

```
// count is the difference_type type defined by vector<int>
vector<int>::difference_type count;
```

These declarations use the scope operator (§ 1.2, p. 8) to say that we want the `iterator` member of the `list<string>` class and the `difference_type` defined by `vector<int>`, respectively.

Exercises Section 9.2.2

Exercise 9.7: What type should be used as the index into a `vector` of `int`s?

Exercise 9.8: What type should be used to read elements in a `list` of `string`s? To write them?

9.2.3. `begin` and `end` Members

The `begin` and `end` operations (§ 3.4.1, p. 106) yield iterators that refer to the first and one past the last element in the container. These iterators are most often used to form an iterator range that encompasses all the elements in the container.

As shown in Table 9.2 (p. 330), there are several versions of `begin` and `end`: The versions with an `r` return reverse iterators (which we cover in § 10.4.3 (p. 407)). Those that start with a `c` return the `const` version of the related iterator:

Click here to view code image

```
list<string> a = {"Milton", "Shakespeare", "Austen"};
auto it1 = a.begin();  // list<string>::iterator
auto it2 = a.rbegin(); // list<string>::reverse_iterator
auto it3 = a.cbegin(); // list<string>::const_iterator
auto it4 = a.crbegin();// list<string>::const_reverse_iterator
```

The functions that do not begin with a `c` are overloaded. That is, there are actually two members named `begin`. One is a `const` member (§ 7.1.2, p. 258) that returns the container's `const_iterator` type. The other is nonconst and returns the container's `iterator` type. Similarly for `rbegin`, `end`, and `rend`. When we call one of these members on a nonconst object, we get the version that returns `iterator`. We get a `const` version of the iterators *only* when we call these functions on a `const` object. As with pointers and references to `const`, we can convert a plain `iterator` to the corresponding `const_iterator`, but not vice versa.

The `c` versions were introduced by the new standard to support using `auto` with `begin` and `end` functions (§ 2.5.2, p. 68). In the past, we had no choice but to say which type of iterator we want:

Click here to view code image

```
// type is explicitly specified
list<string>::iterator it5 = a.begin();
list<string>::const_iterator it6 = a.begin();
// iterator or const_iterator depending on a's type of a
auto it7 = a.begin();  // const_iterator only if a is const
auto it8 = a.cbegin(); // it8 is const_iterator
```

When we use `auto` with `begin` or `end`, the iterator type we get depends on the container type. How we intend to use the iterator is irrelevant. The `c` versions let us get a `const_iterator` regardless of the type of the container.

Best Practices

When write access is not needed, use `cbegin` and `cend`.

Exercises Section 9.2.3

Exercise 9.9: What is the difference between the `begin` and `cbegin` functions?

Exercise 9.10: What are the types of the following four objects?

Click here to view code image

```
vector<int> v1;
const vector<int> v2;
auto it1 = v1.begin(), it2 = v2.begin();
auto it3 = v1.cbegin(), it4 = v2.cbegin();
```

9.2.4. Defining and Initializing a Container

Every container type defines a default constructor (§ 7.1.4, p. 263). With the exception of `array`, the default constructor creates an empty container of the specified type. Again excepting `array`, the other constructors take arguments that specify the size of the container and initial values for the elements.

Initializing a Container as a Copy of Another Container

There are two ways to create a new container as a copy of another one: We can directly copy the container, or (excepting `array`) we can copy a range of elements

denoted by a pair of iterators.

To create a container as a copy of another container, the container and element types must match. When we pass iterators, there is no requirement that the container types be identical. Moreover, the element types in the new and original containers can differ as long as it is possible to convert (§ 4.11, p. 159) the elements we're copying to the element type of the container we are initializing:

Click here to view code image

```
// each container has three elements, initialized from the given initializers
list<string> authors = {"Milton", "Shakespeare", "Austen"};
vector<const char*> articles = {"a", "an", "the"};
list<string> list2(authors);      // ok: types match
deque<string> authList(authors); // error: container types don't match
vector<string> words(articles);  // error: element types must match
// ok: converts const char* elements to string
forward_list<string> words(articles.begin(), articles.end());
```

Note

When we initialize a container as a copy of another container, the container type and element type of both containers must be identical.

The constructor that takes two iterators uses them to denote a range of elements that we want to copy. As usual, the iterators mark the first and one past the last element to be copied. The new container has the same size as the number of elements in the range. Each element in the new container is initialized by the value of the corresponding element in the range.

Because the iterators denote a range, we can use this constructor to copy a subsequence of a container. For example, assuming `it` is an iterator denoting an element in `authors`, we can write

Click here to view code image

```
// copies up to but not including the element denoted by it
deque<string> authList(authors.begin(), it);
```

List Initialization

Under the new standard, we can list initialize (§ 3.3.1, p. 98) a container:

Click here to view code image

```
// each container has three elements, initialized from the given initializers
```

```
list<string> authors = {"Milton", "Shakespeare", "Austen"};
vector<const char*> articles = {"a", "an", "the"};
```

When we do so, we explicitly specify values for each element in the container. For types other than `array`, the initializer list also implicitly specifies the size of the container: The container will have as many elements as there are initializers.

Sequential Container Size-Related Constructors

In addition to the constructors that sequential containers have in common with associative containers, we can also initialize the sequential containers (other than `array`) from a size and an (optional) element initializer. If we do not supply an element initializer, the library creates a value-initialized one for us § 3.3.1 (p. 98):

Click here to view code image

```
vector<int> ivec(10, -1);          // ten int elements, each initialized to
-1
list<string> svec(10, "hi!");      // ten strings; each element is "hi!"
forward_list<int> ivec(10);        // ten elements, each initialized to 0
deque<string> svec(10);            // ten elements, each an empty string
```

We can use the constructor that takes a size argument if the element type is a built-in type or a class type that has a default constructor (§ 9.2, p. 329). If the element type does not have a default constructor, then we must specify an explicit element initializer along with the size.

Note

The constructors that take a size are valid *only* for sequential containers; they are not supported for the associative containers.

Library arrays Have Fixed Size

Just as the size of a built-in array is part of its type, the size of a library `array` is part of its type. When we define an `array`, in addition to specifying the element type, we also specify the container size:

Click here to view code image

```
array<int, 42>      // type is: array that holds 42 ints
array<string, 10> // type is: array that holds 10 strings
```

To use an `array` type we must specify both the element type and the size:

Click here to view code image

```
array<int, 10>::size_type i; // array type includes element type and size
array<int>::size_type j;     // error: array<int> is not a type
```

Because the size is part of the `array`'s type, `array` does not support the normal container constructors. Those constructors, implicitly or explicitly, determine the size of the container. It would be redundant (at best) and error-prone to allow users to pass a size argument to an `array` constructor.

The fixed-size nature of `arrays` also affects the behavior of the constructors that `array` does define. Unlike the other containers, a default-constructed `array` is not empty: It has as many elements as its size. These elements are default initialized (§ 2.2.1, p. 43) just as are elements in a built-in array (§ 3.5.1, p. 114). If we list initialize the `array`, the number of the initializers must be equal to or less than the size of the `array`. If there are fewer initializers than the size of the `array`, the initializers are used for the first elements and any remaining elements are value initialized (§ 3.3.1, p. 98). In both cases, if the element type is a class type, the class must have a default constructor in order to permit value initialization:

Click here to view code image

```
array<int, 10> ia1;  // ten default-initialized ints
array<int, 10> ia2 = {0,1,2,3,4,5,6,7,8,9};  // list initialization
array<int, 10> ia3 = {42};  // ia3[0] is 42, remaining elements are 0
```

It is worth noting that although we cannot copy or assign objects of built-in array types (§ 3.5.1, p. 114), there is no such restriction on `array`:

Click here to view code image

```
int digs[10] = {0,1,2,3,4,5,6,7,8,9};
int cpy[10] = digs;  // error: no copy or assignment for built-in arrays
array<int, 10> digits = {0,1,2,3,4,5,6,7,8,9};
array<int, 10> copy = digits;  // ok: so long as array types match
```

As with any container, the initializer must have the same type as the container we are creating. For `arrays`, the element type and the size must be the same, because the size of an `array` is part of its type.

Exercises Section 9.2.4

Exercise 9.11: Show an example of each of the six ways to create and initialize a `vector`. Explain what values each `vector` contains.

Exercise 9.12: Explain the differences between the constructor that takes a container to copy and the constructor that takes two iterators.

Exercise 9.13: How would you initialize a `vector<double>` from a `list<int>`? From a `vector<int>`? Write code to check your answers.

9.2.5. Assignment and swap

The assignment-related operators, listed in Table 9.4 (overleaf) act on the entire container. The assignment operator replaces the entire range of elements in the left-hand container with copies of the elements from the right-hand operand:

Click here to view code image

```
c1 = c2;        // replace the contents of c1 with a copy of the elements in c2
c1 = {a,b,c}; // after the assignment c1 has size 3
```

Table 9.4. Container Assignment Operations

`c1 = c2`	Replace the elements in `c1` with copies of the elements in `c2`. `c1` and `c2` must be the same type.
`c = {a,b,c...}`	Replace the elements in `c1` with copies of the elements in the initializer list. **(Not valid for `array`.)**
`swap(c1, c2)` `c1.swap(c2)`	Exchanges elements in `c1` with those in `c2`. `c1` and `c2` must be the same type. `swap` is usually *much* faster than copying elements from `c2` to `c1`.
`assign` operations not valid for associative containers or `array`	
`seq.assign(b,e)`	Replaces elements in `seq` with those in the range denoted by iterators `b` and `e`. The iterators `b` and `e` must not refer to elements in `seq`.
`seq.assign(il)`	Replaces the elements in `seq` with those in the initializer list `il`.
`seq.assign(n,t)`	Replaces the elements in `seq` with `n` elements with value `t`.

Assignment related operations invalidate iterators, references, and pointers into the left-hand container. Aside from `string` they remain valid after a `swap`, and (excepting `arrays`) the containers to which they refer are swapped.

After the first assignment, the left- and right-hand containers are equal. If the containers had been of unequal size, after the assignment both containers would have the size of the right-hand operand. After the second assignment, the `size` of `c1` is 3, which is the number of values provided in the braced list.

Unlike built-in arrays, the library `array` type does allow assignment. The left-and right-hand operands must have the same type:

Click here to view code image

```
array<int, 10> a1 = {0,1,2,3,4,5,6,7,8,9};
array<int, 10> a2 = {0}; // elements all have value 0
a1 = a2;   // replaces elements in a1
a2 = {0}; // error: cannot assign to an array from a braced list
```

Because the size of the right-hand operand might differ from the size of the left-hand operand, the `array` type does not support `assign` and it does not allow assignment from a braced list of values.

Using assign (Sequential Containers Only)

The assignment operator requires that the left-hand and right-hand operands have the same type. It copies all the elements from the right-hand operand into the left-hand operand. The sequential containers (except `array`) also define a member named `assign` that lets us assign from a different but compatible type, or assign from a subsequence of a container. The `assign` operation replaces all the elements in the left-hand container with (copies of) the elements specified by its arguments. For example, we can use `assign` to assign a range of `char*` values from a `vector` into a `list` of `string`:

Click here to view code image

```
list<string> names;
vector<const char*> oldstyle;
names = oldstyle;   // error: container types don't match
// ok: can convert from const char*to string
names.assign(oldstyle.cbegin(), oldstyle.cend());
```

The call to `assign` replaces the elements in `names` with copies of the elements in the range denoted by the iterators. The arguments to `assign` determine how many elements and what values the container will have.

> **Warning**
>
> Because the existing elements are replaced, the iterators passed to `assign` must not refer to the container on which `assign` is called.

A second version of `assign` takes an integral value and an element value. It replaces the elements in the container with the specified number of elements, each of which has the specified element value:

Click here to view code image

```
// equivalent to slist1.clear();
// followed by slist1.insert(slist1.begin(), 10, "Hiya!");
list<string> slist1(1);       // one element, which is the empty string
slist1.assign(10, "Hiya!"); // ten elements; each one is Hiya !
```

Using swap

The `swap` operation exchanges the contents of two containers of the same type. After the call to `swap`, the elements in the two containers are interchanged:

Click here to view code image

```
vector<string> svec1(10); // vector with ten elements
vector<string> svec2(24); // vector with 24 elements
swap(svec1, svec2);
```

After the swap, svec1 contains 24 string elements and svec2 contains ten. With the exception of arrays, swapping two containers is guaranteed to be fast—the elements themselves are not swapped; internal data structures are swapped.

Note

Excepting array, swap does not copy, delete, or insert any elements and is guaranteed to run in constant time.

The fact that elements are not moved means that, with the exception of string, iterators, references, and pointers into the containers are not invalidated. They refer to the same elements as they did before the swap. However, after the swap, those elements are in a different container. For example, had iter denoted the string at position svec1 [3] before the swap, it will denote the element at position svec2[3] after the swap. Differently from the containers, a call to swap on a string may invalidate iterators, references and pointers.

Unlike how swap behaves for the other containers, swapping two arrays does exchange the elements. As a result, swapping two arrays requires time proportional to the number of elements in the array.

After the swap, pointers, references, and iterators remain bound to the same element they denoted before the swap. Of course, the value of that element has been swapped with the corresponding element in the other array.

In the new library, the containers offer both a member and nonmember version of swap. Earlier versions of the library defined only the member version of swap. The nonmember swap is of most importance in generic programs. As a matter of habit, it is best to use the nonmember version of swap.

Exercises Section 9.2.5

Exercise 9.14: Write a program to assign the elements from a list of char* pointers to C-style character strings to a vector of strings.

9.2.6. Container Size Operations

With one exception, the container types have three size-related operations. The `size` member (§ 3.2.2, p. 87) returns the number of elements in the container; `empty` returns a `bool` that is `true` if `size` is zero and `false` otherwise; and `max_size` returns a number that is greater than or equal to the number of elements a container of that type can contain. For reasons we'll explain in the next section, `forward_list` provides `max_size` and `empty`, but not `size`.

9.2.7. Relational Operators

Every container type supports the equality operators (== and !=); all the containers except the unordered associative containers also support the relational operators (>, >=, <, <=). The right- and left-hand operands must be the same kind of container and must hold elements of the same type. That is, we can compare a `vector<int>` only with another `vector<int>`. We cannot compare a `vector<int>` with a `list<int>` or a `vector<double>`.

Comparing two containers performs a pairwise comparison of the elements. These operators work similarly to the `string` relationals (§ 3.2.2, p. 88):

- If both containers are the same size and all the elements are equal, then the two containers are equal; otherwise, they are unequal.
- If the containers have different sizes but every element of the smaller one is equal to the corresponding element of the larger one, then the smaller one is less than the other.
- If neither container is an initial subsequence of the other, then the comparison depends on comparing the first unequal elements.

The following examples illustrate how these operators work:

Click here to view code image

```
vector<int> v1 = { 1, 3, 5, 7, 9, 12 };
vector<int> v2 = { 1, 3, 9 };
vector<int> v3 = { 1, 3, 5, 7 };
vector<int> v4 = { 1, 3, 5, 7, 9, 12 };
v1 < v2  // true; v1 and v2 differ at element [2]: v1[2] is less than v2[2]
v1 < v3  // false; all elements are equal, but v3 has fewer of them;
v1 == v4 // true; each element is equal and v1 and v4 have the same size()
v1 == v2 // false; v2 has fewer elements than v1
```

Relational Operators Use Their Element's Relational Operator

Note

We can use a relational operator to compare two containers only if the appropriate comparison operator is defined for the element type.

The container equality operators use the element's == operator, and the relational operators use the element's < operator. If the element type doesn't support the required operator, then we cannot use the corresponding operations on containers holding that type. For example, the `Sales_data` type that we defined in Chapter 7 does not define either the == or the < operation. Therefore, we cannot compare two containers that hold `Sales_data` elements:

Click here to view code image

```
vector<Sales_data> storeA, storeB;
if (storeA < storeB) // error: Sales_data has no less-than operator
```

Exercises Section 9.2.7

Exercise 9.15: Write a program to determine whether two `vector<int>`s are equal.

Exercise 9.16: Repeat the previous program, but compare elements in a `list<int>` to a `vector<int>`.

Exercise 9.17: Assuming `c1` and `c2` are containers, what (if any) constraints does the following usage place on the types of `c1` and `c2`?

```
if (c1 < c2)
```

9.3. Sequential Container Operations

The sequential and associative containers differ in how they organize their elements. These differences affect how elements are stored, accessed, added, and removed. The previous section covered operations common to all containers (those listed in Table 9.2 (p. 330)). We'll cover the operations specific to the sequential containers in the remainder of this chapter.

9.3.1. Adding Elements to a Sequential Container

Excepting `array`, all of the library containers provide flexible memory management. We can add or remove elements dynamically changing the size of the container at run time. Table 9.5 (p. 343) lists the operations that add elements to a (non`array`) sequential container.

Table 9.5. Operations That Add Elements to a Sequential Container

These operations change the size of the container; they are not supported by `array`. `forward_list` has special versions of `insert` and `emplace`; see § 9.3.4 (p. 350). `push_back` and `emplace_back` not valid for `forward_list`. `push_front` and `emplace_front` not valid for `vector` or `string`.

`c.push_back(t)`	Creates an element with value `t` or constructed from *args* at the end of `c`. Returns `void`.
`c.emplace_back(args)`	
`c.push_front(t)`	Creates an element with value `t` or constructed from *args* on the front of `c`. Returns `void`.
`c.emplace_front(args)`	
`c.insert(p,t)`	Creates an element with value `t` or constructed from *args* before the element denoted by iterator `p`. Returns an iterator referring to the element that was added.
`c.emplace(p, args)`	
`c.insert(p,n,t)`	Inserts `n` elements with value `t` before the element denoted by iterator `p`. Returns an iterator to the first element inserted; if `n` is zero, returns `p`.
`c.insert(p,b,e)`	Inserts the elements from the range denoted by iterators `b` and `e` before the element denoted by iterator `p`. `b` and `e` may not refer to elements in `c`. Returns an iterator to the first element inserted; if the range is empty, returns `p`.
`c.insert(p,il)`	`il` is a braced list of element values. Inserts the given values before the element denoted by the iterator `p`. Returns an iterator to the first inserted element; if the list is empty returns `p`.

Adding elements to a `vector`, `string`, or `deque` potentially invalidates all existing iterators, references, and pointers into the container.

When we use these operations, we must remember that the containers use different strategies for allocating elements and that these strategies affect performance. Adding elements anywhere but at the end of a `vector` or `string`, or anywhere but the beginning or end of a `deque`, requires elements to be moved. Moreover, adding elements to a `vector` or a `string` may cause the entire object to be reallocated. Reallocating an object requires allocating new memory and moving elements from the old space to the new.

Using push_back

In § 3.3.2 (p. 100) we saw that `push_back` appends an element to the back of a `vector`. Aside from `array` and `forward_list`, every sequential container (including the `string` type) supports `push_back`.

As an example, the following loop reads one `string` at a time into `word`:

Click here to view code image

```
// read from standard input, putting each word onto the end of container
string word;
while (cin >> word)
    container.push_back(word);
```

The call to `push_back` creates a new element at the end of `container`, increasing

the `size` of `container` by 1. The value of that element is a copy of `word`. The type of `container` can be any of `list`, `vector`, or `deque`.

Because `string` is just a container of characters, we can use `push_back` to add characters to the end of the `string`:

Click here to view code image

```
void pluralize(size_t cnt, string &word)
{
    if (cnt > 1)
        word.push_back('s');  // same as word += 's'
}
```

> **Key Concept: Container Elements Are Copies**
>
> When we use an object to initialize a container, or insert an object into a container, a copy of that object's value is placed in the container, not the object itself. Just as when we pass an object to a nonreference parameter (§ 6.2.1, p. 209), there is no relationship between the element in the container and the object from which that value originated. Subsequent changes to the element in the container have no effect on the original object, and vice versa.

Using push_front

In addition to `push_back`, the `list`, `forward_list`, and `deque` containers support an analogous operation named `push_front`. This operation inserts a new element at the front of the container:

Click here to view code image

```
list<int> ilist;
// add elements to the start of ilist
for (size_t ix = 0; ix != 4; ++ix)
    ilist.push_front(ix);
```

This loop adds the elements `0`, `1`, `2`, `3` to the beginning of `ilist`. Each element is inserted at the *new beginning* of the `list`. That is, when we insert `1`, it goes in front of `0`, and `2` in front of `1`, and so forth. Thus, the elements added in a loop such as this one wind up in reverse order. After executing this loop, `ilist` holds the sequence `3,2,1,0`.

Note that `deque`, which like `vector` offers fast random access to its elements, provides the `push_front` member even though `vector` does not. A `deque` guarantees constant-time insert and delete of elements at the beginning and end of the container. As with `vector`, inserting elements other than at the front or back of a `deque` is a potentially expensive operation.

Adding Elements at a Specified Point in the Container

The `push_back` and `push_front` operations provide convenient ways to insert a single element at the end or beginning of a sequential container. More generally, the `insert` members let us insert zero or more elements at any point in the container. The `insert` members are supported for `vector`, `deque`, `list`, and `string`. `forward_list` provides specialized versions of these members that we'll cover in § 9.3.4 (p. 350).

Each of the `insert` functions takes an iterator as its first argument. The iterator indicates where in the container to put the element(s). It can refer to any position in the container, including one past the end of the container. Because the iterator might refer to a nonexistent element off the end of the container, and because it is useful to have a way to insert elements at the beginning of a container, element(s) are inserted *before* the position denoted by the iterator. For example, this statement

Click here to view code image

```
slist.insert(iter, "Hello!"); // insert "Hello!" just before iter
```

inserts a `string` with value `"Hello"` just before the element denoted by `iter`.

Even though some containers do not have a `push_front` operation, there is no similar constraint on `insert`. We can `insert` elements at the beginning of a container without worrying about whether the container has `push_front`:

Click here to view code image

```
vector<string> svec;
list<string> slist;
// equivalent to calling slist.push_front("Hello!");
slist.insert(slist.begin(), "Hello!");
// no push_front on vector but we can insert before begin()
// warning: inserting anywhere but at the end of a vector might be slow
svec.insert(svec.begin(), "Hello!");
```

> **Warning**
>
> It is legal to insert anywhere in a `vector`, `deque`, or `string`. However, doing so can be an expensive operation.

Inserting a Range of Elements

The arguments to `insert` that appear after the initial iterator argument are analogous to the container constructors that take the same parameters. The version that takes an element count and a value adds the specified number of identical

elements before the given position:

Click here to view code image

```
svec.insert(svec.end(), 10, "Anna");
```

This code inserts ten elements at the end of `svec` and initializes each of those elements to the `string "Anna"`.

The versions of `insert` that take a pair of iterators or an initializer list insert the elements from the given range before the given position:

Click here to view code image

```
vector<string> v = {"quasi", "simba", "frollo", "scar"};
// insert the last two elements of v at the beginning of slist
slist.insert(slist.begin(), v.end() - 2, v.end());
slist.insert(slist.end(), {"these", "words", "will",
                           "go", "at", "the", "end"});
// run-time error: iterators denoting the range to copy from
// must not refer to the same container as the one we are changing
slist.insert(slist.begin(), slist.begin(), slist.end());
```

When we pass a pair of iterators, those iterators may not refer to the same container as the one to which we are adding elements.

Under the new standard, the versions of `insert` that take a count or a range return an iterator to the first element that was inserted. (In prior versions of the library, these operations returned `void`.) If the range is empty, no elements are inserted, and the operation returns its first parameter.

Using the Return from insert

We can use the value returned by `insert` to repeatedly insert elements at a specified position in the container:

Click here to view code image

```
list<string> lst;
auto iter = lst.begin();
while (cin >> word)
   iter = lst.insert(iter, word); // same as calling push_front
```

Note

It is important to understand how this loop operates—in particular, to understand why the loop is equivalent to calling `push_front`.

Before the loop, we initialize `iter` to `lst.begin()`. The first call to `insert` takes the `string` we just read and puts it in front of the element denoted by `iter`. The value returned by `insert` is an iterator referring to this new element. We assign that iterator to `iter` and repeat the `while`, reading another word. As long as there are words to insert, each trip through the `while` inserts a new element ahead of `iter` and reassigns to `iter` the location of the newly inserted element. That element is the (new) first element. Thus, each iteration inserts an element ahead of the first element in the `list`.

Using the Emplace Operations

The new standard introduced three new members—`emplace_front`, `emplace`, and `emplace_back`—that construct rather than copy elements. These operations correspond to the `push_front`, `insert`, and `push_back` operations in that they let us put an element at the front of the container, in front of a given position, or at the back of the container, respectively.

When we call a push or insert member, we pass objects of the element type and those objects are copied into the container. When we call an emplace member, we pass arguments to a constructor for the element type. The emplace members use those arguments to construct an element directly in space managed by the container. For example, assuming `c` holds `Sales_data` (§ 7.1.4, p. 264) elements:

Click here to view code image

```
// construct a Sales_data object at the end of c
// uses the three-argument Sales_data constructor
c.emplace_back("978-0590353403", 25, 15.99);
// error: there is no version of push_back that takes three arguments
c.push_back("978-0590353403", 25, 15.99);
// ok: we create a temporary Sales_data object to pass to push_back
c.push_back(Sales_data("978-0590353403", 25, 15.99));
```

The call to `emplace_back` and the second call to `push_back` both create new `Sales_data` objects. In the call to `emplace_back`, that object is created directly in space managed by the container. The call to `push_back` creates a local temporary object that is pushed onto the container.

The arguments to an emplace function vary depending on the element type. The arguments must match a constructor for the element type:

Click here to view code image

```
// iter refers to an element in c, which holds Sales_data elements
c.emplace_back(); // uses the Sales_data default constructor
c.emplace(iter, "999-999999999"); // uses Sales_data(string)
// uses the Sales_data constructor that takes an ISBN, a count, and a price
```

```
c.emplace_front("978-0590353403", 25, 15.99);
```

Note

The *emplace* functions construct elements in the container. The arguments to these functions must match a constructor for the element type.

Exercises Section 9.3.1

Exercise 9.18: Write a program to read a sequence of `strings` from the standard input into a `deque`. Use iterators to write a loop to print the elements in the `deque`.

Exercise 9.19: Rewrite the program from the previous exercise to use a `list`. List the changes you needed to make.

Exercise 9.20: Write a program to copy elements from a `list<int>` into two `deques`. The even-valued elements should go into one `deque` and the odd ones into the other.

Exercise 9.21: Explain how the loop from page 345 that used the return from `insert` to add elements to a `list` would work if we inserted into a `vector` instead.

Exercise 9.22: Assuming `iv` is a `vector` of `ints`, what is wrong with the following program? How might you correct the problem(s)?

Click here to view code image

```
vector<int>::iterator iter = iv.begin(),
                      mid = iv.begin() + iv.size()/2;
while (iter != mid)
    if (*iter == some_val)
        iv.insert(iter, 2 * some_val);
```

9.3.2. Accessing Elements

Table 9.6 lists the operations we can use to access elements in a sequential container. The access operations are undefined if the container has no elements.

Table 9.6. Operations to Access Elements in a Sequential Container

at and subscript operator valid only for `string`, `vector`, `deque`, and `array`. `back` not valid for `forward_list`.

`c.back()`	Returns a reference to the last element in `c`. Undefined if `c` is empty.
`c.front()`	Returns a reference to the first element in `c`. Undefined if `c` is empty.
`c[n]`	Returns a reference to the element indexed by the unsigned integral value `n`. Undefined if `n >= c.size()`.
`c.at(n)`	Returns a reference to the element indexed by `n`. If the index is out of range, throws an `out_of_range` exception.

Calling `front` or `back` on an empty container, like using a subscript that is out of range, is a serious programming error.

Each sequential container, including `array`, has a `front` member, and all except `forward_list` also have a `back` member. These operations return a reference to the first and last element, respectively:

Click here to view code image

```
// check that there are elements before dereferencing an iterator or calling front or back
if (!c.empty()) {
    // val and val2 are copies of the value of the first element in c
    auto val = *c.begin(), val2 = c.front();
    // val3 and val4 are copies of the of the last element in c
    auto last = c.end();
    auto val3 = *(--last); // can't decrement forward_list iterators
    auto val4 = c.back();  // not supported by forward_list
}
```

This program obtains references to the first and last elements in `c` in two different ways. The direct approach is to call `front` or `back`. Indirectly, we can obtain a reference to the same element by dereferencing the iterator returned by `begin` or decrementing and then dereferencing the iterator returned by `end`.

Two things are noteworthy in this program: The `end` iterator refers to the (nonexistent) element one past the end of the container. To fetch the last element we must first decrement that iterator. The other important point is that before calling `front` or `back` (or dereferencing the iterators from `begin` or `end`), we check that `c` isn't empty. If the container were empty, the operations inside the `if` would be undefined.

The Access Members Return References

The members that access elements in a container (i.e., `front`, `back`, subscript, and `at`) return references. If the container is a `const` object, the return is a reference to `const`. If the container is not `const`, the return is an ordinary reference that we can use to change the value of the fetched element:

Click here to view code image

```
if (!c.empty()) {
    c.front()  = 42;       // assigns 42 to the first element in c
    auto &v =  c.back();   // get a reference to the last element
    v = 1024;              // changes the element in c
    auto v2 =  c.back();   // v2 is not a reference; it's a copy of c.back()
    v2 = 0;                // no change to the element in c
}
```

As usual, if we use `auto` to store the return from one of these functions and we want to use that variable to change the element, we must remember to define our variable as a reference type.

Subscripting and Safe Random Access

The containers that provide fast random access (`string`, `vector`, `deque`, and `array`) also provide the subscript operator (§ 3.3.3, p. 102). As we've seen, the subscript operator takes an index and returns a reference to the element at that position in the container. The index must be "in range," (i.e., greater than or equal to 0 and less than the size of the container). It is up to the program to ensure that the index is valid; the subscript operator does not check whether the index is in range. Using an out-of-range value for an index is a serious programming error, but one that the compiler will not detect.

If we want to ensure that our index is valid, we can use the `at` member instead. The `at` member acts like the subscript operator, but if the index is invalid, `at` throws an `out_of_range` exception (§ 5.6, p. 193):

Click here to view code image

```
vector<string> svec; // empty vector
cout << svec[0];     // run-time error: there are no elements in svec!
cout << svec.at(0);  // throws an out_of_range exception
```

Exercises Section 9.3.2

Exercise 9.23: In the first program in this section on page 346, what would the values of `val`, `val2`, `val3`, and `val4` be if `c.size()` is `1`?

Exercise 9.24: Write a program that fetches the first element in a `vector` using `at`, the subscript operator, `front`, and `begin`. Test your program on an empty `vector`.

9.3.3. Erasing Elements

Just as there are several ways to add elements to a (nonarray) container there are also several ways to remove elements. These members are listed in Table 9.7.

Table 9.7. erase Operations on Sequential Containers

These operations change the size of the container and so are not supported by `array`. `forward_list` has a special version of `erase`; see § 9.3.4 (p. 350). `pop_back` not valid for `forward_list`; `pop_front` not valid for `vector` and `string`.

`c.pop_back()`	Removes last element in `c`. Undefined if `c` is empty. Returns `void`.
`c.pop_front()`	Removes first element in `c`. Undefined if `c` is empty. Returns `void`.
`c.erase(p)`	Removes the element denoted by the iterator `p` and returns an iterator to the element after the one deleted or the off-the-end iterator if `p` denotes the last element. Undefined if `p` is the off-the-end iterator.
`c.erase(b,e)`	Removes the range of elements denoted by the iterators `b` and `e`. Returns an iterator to the element after the last one that was deleted, or an off-the-end iterator if `e` is itself an off-the-end iterator.
`c.clear()`	Removes all the elements in `c`. Returns `void`.

Removing elements anywhere but the beginning or end of a `deque` invalidates all iterators, references, and pointers. Iterators, references, and pointers to elements after the erasure point in a `vector` or `string` are invalidated.

Warning

The members that remove elements do not check their argument(s). The programmer must ensure that element(s) exist before removing them.

The pop_front and pop_back Members

The `pop_front` and `pop_back` functions remove the first and last elements, respectively. Just as there is no `push_front` for `vector` and `string`, there is also no `pop_front` for those types. Similarly, `forward_list` does not have `pop_back`. Like the element access members, we may not use a pop operation on an empty container.

These operations return `void`. If you need the value you are about to pop, you must store that value before doing the pop:

Click here to view code image

```
while (!ilist.empty()) {
    process(ilist.front()); // do something with the current top of ilist
    ilist.pop_front();      // done; remove the first element
```

```
}
```

Removing an Element from within the Container

The `erase` members remove element(s) at a specified point in the container. We can delete a single element denoted by an iterator or a range of elements marked by a pair of iterators. Both forms of `erase` return an iterator referring to the location *after* the (last) element that was removed. That is, if `j` is the element following `i`, then `erase(i)` will return an iterator referring to `j`.

As an example, the following loop erases the odd elements in a `list`:

Click here to view code image

```
list<int> lst = {0,1,2,3,4,5,6,7,8,9};
auto it = lst.begin();
while (it != lst.end())
    if (*it % 2)              // if the element is odd
        it = lst.erase(it);   // erase this element
    else
        ++it;
```

On each iteration, we check whether the current element is odd. If so, we `erase` that element, setting `it` to denote the element after the one we erased. If `*it` is even, we increment `it` so we'll look at the next element on the next iteration.

Removing Multiple Elements

The iterator-pair version of `erase` lets us delete a range of elements:

Click here to view code image

```
// delete the range of elements between two iterators
// returns an iterator to the element just after the last removed element
elem1 = slist.erase(elem1, elem2); // after the call elem1 == elem2
```

The iterator `elem1` refers to the first element we want to erase, and `elem2` refers to one past the last element we want to remove.

To delete all the elements in a container, we can either call `clear` or pass the iterators from `begin` and `end` to `erase`:

Click here to view code image

```
slist.clear(); // delete all the elements within the container
slist.erase(slist.begin(), slist.end()); // equivalent
```

Exercises Section 9.3.3

Exercise 9.25: In the program on page 349 that erased a range of

elements, what happens if `elem1` and `elem2` are equal? What if `elem2` or both `elem1` and `elem2` are the off-the-end iterator?

Exercise 9.26: Using the following definition of `ia`, copy `ia` into a `vector` and into a `list`. Use the single-iterator form of `erase` to remove the elements with odd values from your `list` and the even values from your `vector`.

Click here to view code image

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 };
```

9.3.4. Specialized forward_list Operations

To understand why `forward_list` has special versions of the operations to add and remove elements, consider what must happen when we remove an element from a singly linked list. As illustrated in Figure 9.1, removing an element changes the links in the sequence. In this case, removing $elem_3$ changes $elem_2$; $elem_2$ had pointed to $elem_3$, but after we remove $elem_3$, $elem_2$ points to $elem_4$.

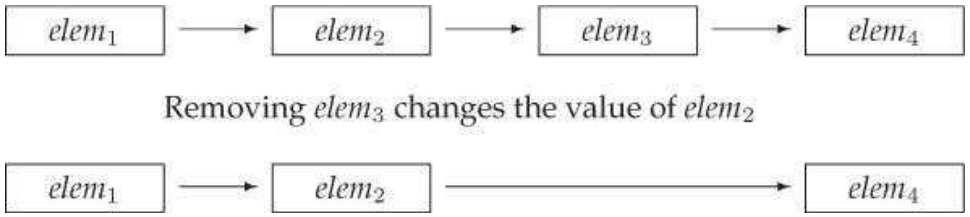


Figure 9.1. forward_list Specialized Operations

When we add or remove an element, the element before the one we added or removed has a different successor. To add or remove an element, we need access to its predecessor in order to update that element's links. However, `forward_list` is a singly linked list. In a singly linked list there is no easy way to get to an element's predecessor. For this reason, the operations to add or remove elements in a `forward_list` operate by changing the element *after* the given element. That way, we always have access to the elements that are affected by the change.

Because these operations behave differently from the operations on the other containers, `forward_list` does not define `insert`, `emplace`, or `erase`. Instead it defines members (listed in Table 9.8) named `insert_after`, `emplace_after`, and `erase_after`. For example, in our illustration, to remove $elem_3$, we'd call `erase_after` on an iterator that denoted $elem_2$. To support these operations, `forward_list` also defines `before_begin`, which returns an **off-the-beginning** iterator. This iterator lets us add or remove elements "after" the nonexistent element before the first one in the list.

Table 9.8. Operations to Insert or Remove Elements in a forward_list

`lst.before_begin()` `lst.cbefore_begin()`	Iterator denoting the nonexistent element just before the beginning of the list. This iterator may not be dereferenced. `cbefore_begin()` returns a `const_iterator`.

`lst.insert_after(p,t)`
`lst.insert_after(p,n,t)`
`lst.insert_after(p,b,e)`
`lst.insert_after(p,il)`
	Inserts element(s) *after* the one denoted by iterator `p`. `t` is an object, `n` is a count, `b` and `e` are iterators denoting a range (`b` and `e` must not refer to `lst`), and `il` is a braced list. Returns an iterator to the *last* inserted element. If the range is empty, returns `p`. Undefined if `p` is the off-the-end iterator.

`emplace_after(p, args)`	Uses *args* to construct an element after the one denoted by iterator `p`. Returns an iterator to the new element. Undefined if `p` is the off-the-end iterator.
`lst.erase_after(p)` `lst.erase_after(b,e)`	Removes the element *after* the one denoted by iterator `p` or the range of elements from the one *after* the iterator `b` up to but not including the one denoted by `e`. Returns an iterator to the element after the one deleted, or the off-the-end iterator if there is no such element. Undefined if `p` denotes the last element in `lst` or is the off-the-end iterator.

When we add or remove elements in a `forward_list`, we have to keep track of two iterators—one to the element we're checking and one to that element's predecessor. As an example, we'll rewrite the loop from page 349 that removed the odd-valued elements from a `list` to use a `forward_list`:

Click here to view code image

```
forward_list<int> flst = {0,1,2,3,4,5,6,7,8,9};
auto prev = flst.before_begin(); // denotes element "off the start" of flst
auto curr = flst.begin();        // denotes the first element in flst
while (curr != flst.end()) {     // while there are still elements to process
    if (*curr % 2)                       // if the element is odd
        curr = flst.erase_after(prev); // erase it and move curr
    else {
        prev = curr;                 // move the iterators to denote the next
        ++curr;                      // element and one before the next element
    }
}
```

Here, `curr` denotes the element we're checking, and `prev` denotes the element before `curr`. We call `begin` to initialize `curr`, so that the first iteration checks whether the first element is even or odd. We initialize `prev` from `before_begin`, which returns an iterator to the nonexistent element just before `curr`.

When we find an odd element, we pass `prev` to `erase_after`. This call erases the element after the one denoted by `prev`; that is, it erases the element denoted by

curr. We reset curr to the return from erase_after, which makes curr denote the next element in the sequence and we leave prev unchanged; prev still denotes the element before the (new) value of curr. If the element denoted by curr is not odd, then we have to move both iterators, which we do in the else.

Exercises Section 9.3.4

Exercise 9.27: Write a program to find and remove the odd-valued elements in a forward_list<int>.

Exercise 9.28: Write a function that takes a forward_list<string> and two additional string arguments. The function should find the first string and insert the second immediately following the first. If the first string is not found, then insert the second string at the end of the list.

9.3.5. Resizing a Container

With the usual exception of arrays, we can use resize, described in Table 9.9, to make a container larger or smaller. If the current size is greater than the requested size, elements are deleted from the back of the container; if the current size is less than the new size, elements are added to the back of the container:

Click here to view code image

```
list<int> ilist(10, 42); // ten ints: each has value 42
ilist.resize(15);        // adds five elements of value 0 to the back of ilist
ilist.resize(25, -1);    // adds ten elements of value -1 to the back of ilist
ilist.resize(5);         // erases 20 elements from the back of ilist
```

Table 9.9. Sequential Container Size Operations

resize not valid for array.	
c.resize(n)	Resize c so that it has n elements. If n < c.size(), the excess elements are discarded. If new elements must be added, they are value initialized.
c.resize(n,t)	Resize c to have n elements. Any elements added have value t.

If resize shrinks the container, then iterators, references, and pointers to the deleted elements are invalidated; resize on a vector, string, or deque potentially invalidates all iterators, pointers, and references.

The resize operation takes an optional element-value argument that it uses to initialize any elements that are added to the container. If this argument is absent, added elements are value initialized (§ 3.3.1, p. 98). If the container holds elements of a class type and resize adds elements, we must supply an initializer or the element type must have a default constructor.

Exercises Section 9.3.5

Exercise 9.29: Given that `vec` holds 25 elements, what does `vec.resize(100)` do? What if we next wrote `vec.resize(10)`?

Exercise 9.30: What, if any, restrictions does using the version of `resize` that takes a single argument place on the element type?

9.3.6. Container Operations May Invalidate Iterators

Operations that add or remove elements from a container can invalidate pointers, references, or iterators to container elements. An invalidated pointer, reference, or iterator is one that no longer denotes an element. Using an invalidated pointer, reference, or iterator is a serious programming error that is likely to lead to the same kinds of problems as using an uninitialized pointer (§ 2.3.2, p. 54).

After an operation that adds elements to a container

- Iterators, pointers, and references to a `vector` or `string` are invalid if the container was reallocated. If no reallocation happens, indirect references to elements before the insertion remain valid; those to elements after the insertion are invalid.
- Iterators, pointers, and references to a `deque` are invalid if we add elements anywhere but at the front or back. If we add at the front or back, iterators are invalidated, but references and pointers to existing elements are not.
- Iterators, pointers, and references (including the off-the-end and the before-the-beginning iterators) to a `list` or `forward_list` remain valid,

It should not be surprising that when we remove elements from a container, iterators, pointers, and references to the removed elements are invalidated. After all, those elements have been destroyed. After we remove an element,

- All other iterators, references, or pointers (including the off-the-end and the before-the-beginning iterators) to a `list` or `forward_list` remain valid.
- All other iterators, references, or pointers to a `deque` are invalidated if the removed elements are anywhere but the front or back. If we remove elements at the back of the `deque`, the off-the-end iterator is invalidated but other iterators, references, and pointers are unaffected; they are also unaffected if we remove from the front.
- All other iterators, references, or pointers to a `vector` or `string` remain valid for elements before the removal point. Note: The off-the-end iterator is always invalidated when we remove elements.

Warning

It is a serious run-time error to use an iterator, pointer, or reference that has been invalidated.

Advice: Managing Iterators

When you use an iterator (or a reference or pointer to a container element), it is a good idea to minimize the part of the program during which an iterator must stay valid.

Because code that adds or removes elements to a container can invalidate iterators, you need to ensure that the iterator is repositioned, as appropriate, after each operation that changes the container. This advice is especially important for `vector`, `string`, and `deque`.

Writing Loops That Change a Container

Loops that add or remove elements of a `vector`, `string`, or `deque` must cater to the fact that iterators, references, or pointers might be invalidated. The program must ensure that the iterator, reference, or pointer is refreshed on each trip through the loop. Refreshing an iterator is easy if the loop calls `insert` or `erase`. Those operations return iterators, which we can use to reset the iterator:

Click here to view code image

```
// silly loop to remove even-valued elements and insert a duplicate of odd-valued elements
vector<int> vi = {0,1,2,3,4,5,6,7,8,9};
auto iter = vi.begin(); // call begin, not cbegin because we're changing vi
while (iter != vi.end()) {
    if (*iter % 2) {
        iter = vi.insert(iter, *iter);  // duplicate the current element
        iter += 2; // advance past this element and the one inserted before it
    } else
        iter = vi.erase(iter);          // remove even elements
        // don't advance the iterator; iter denotes the element after the one we erased
}
```

This program removes the even-valued elements and duplicates each odd-valued one. We refresh the iterator after both the `insert` and the `erase` because either operation can invalidate the iterator.

After the call to `erase`, there is no need to increment the iterator, because the iterator returned from `erase` denotes the next element in the sequence. After the call to `insert`, we increment the iterator twice. Remember, `insert` inserts *before* the position it is given and returns an iterator to the inserted element. Thus, after calling `insert`, `iter` denotes the (newly added) element in front of the one we are processing. We add two to skip over the element we added and the one we just processed. Doing so positions the iterator on the next, unprocessed element.

Avoid Storing the Iterator Returned from end

When we add or remove elements in a `vector` or `string`, or add elements or remove any but the first element in a `deque`, the iterator returned by `end` is *always* invalidated. Thus, loops that add or remove elements should always call `end` rather than use a stored copy. Partly for this reason, C++ standard libraries are usually implemented so that calling `end()` is a very fast operation.

As an example, consider a loop that processes each element and adds a new element following the original. We want the loop to ignore the added elements, and to process only the original elements. After each insertion, we'll position the iterator to denote the next original element. If we attempt to "optimize" the loop, by storing the iterator returned by `end()`, we'll have a disaster:

Click here to view code image

```
// disaster: the behavior of this loop is undefined
auto begin = v.begin(),
     end = v.end(); // bad idea, saving the value of the end iterator
while (begin != end) {
    // do some processing
    // insert the new value and reassign begin, which otherwise would be invalid
    ++begin;  // advance begin because we want to insert after this element
    begin = v.insert(begin, 42);  // insert the new value
    ++begin;  // advance begin past the element we just added
}
```

The behavior of this code is undefined. On many implementations, we'll get an infinite loop. The problem is that we stored the value returned by the `end` operation in a local variable named `end`. In the body of the loop, we added an element. Adding an element invalidates the iterator stored in `end`. That iterator neither refers to an element in `v` nor any longer refers to one past the last element in `v`.

Tip

Don't cache the iterator returned from `end()` in loops that insert or delete elements in a `deque`, `string`, or `vector`.

Rather than storing the `end()` iterator, we must recompute it after each insertion:

Click here to view code image

```
// safer: recalculate end on each trip whenever the loop adds/erases elements
while (begin != v.end()) {
    // do some processing
    ++begin;  // advance begin because we want to insert after this element
    begin = v.insert(begin, 42);  // insert the new value
    ++begin;  // advance begin past the element we just added
}
```

9.4. How a vector Grows

To support fast random access, `vector` elements are stored contiguously—each element is adjacent to the previous element. Ordinarily, we should not care about how a library type is implemented; all we should care about is how to use it. However, in the case of `vectors` and `strings`, part of the implementation leaks into its interface.

Given that elements are contiguous, and that the size of the container is flexible, consider what must happen when we add an element to a `vector` or a `string`: If there is no room for the new element, the container can't just add an element somewhere else in memory—the elements must be contiguous. Instead, the container must allocate new memory to hold the existing elements plus the new one, move the elements from the old location into the new space, add the new element, and deallocate the old memory. If `vector` did this memory allocation and deallocation each time we added an element, performance would be unacceptably slow.

Exercises Section 9.3.6

Exercise 9.31: The program on page 354 to remove even-valued elements and duplicate odd ones will not work on a `list` or `forward_list`. Why? Revise the program so that it works on these types as well.

Exercise 9.32: In the program onpage 354 would it be legal to write the call to `insert` as follows? If not, why not?

Click here to view code image

```
iter = vi.insert(iter, *iter++);
```

Exercise 9.33: In the final example in this section what would happen if we did not assign the result of `insert` to `begin`? Write a program that omits this assignment to see if your expectation was correct.

Exercise 9.34: Assuming `vi` is a container of `int`s that includes even and odd values, predict the behavior of the following loop. After you've analyzed this loop, write a program to test whether your expectations were correct.

Click here to view code image

```
iter = vi.begin();
while (iter != vi.end())
    if (*iter % 2)
        iter = vi.insert(iter, *iter);
    ++iter;
```

To avoid these costs, library implementors use allocation strategies that reduce the number of times the container is reallocated. When they have to get new memory, `vector` and `string` implementations typically allocate capacity beyond what is immediately needed. The container holds this storage in reserve and uses it to allocate new elements as they are added. Thus, there is no need to reallocate the container for each new element.

This allocation strategy is dramatically more efficient than reallocating the container each time an element is added. In fact, its performance is good enough that in practice a `vector` usually grows more efficiently than a `list` or a `deque`, even though the `vector` has to move all of its elements each time it reallocates memory.

Members to Manage Capacity

The `vector` and `string` types provide members, described in Table 9.10, that let us interact with the memory-allocation part of the implementation. The `capacity` operation tells us how many elements the container can hold before it must allocate more space. The `reserve` operation lets us tell the container how many elements it should be prepared to hold.

Table 9.10. Container Size Management

`shrink_to_fit` valid only for `vector`, `string`, and `deque`.
`capacity` and `reserve` valid only for `vector` and `string`.

`c.shrink_to_fit()`	Request to reduce `capacity()` to equal `size()`.
`c.capacity()`	Number of elements `c` can have before reallocation is necessary.
`c.reserve(n)`	Allocate space for at least `n` elements.

Note

> `reserve` does not change the number of elements in the container; it affects only how much memory the `vector` preallocates.

A call to `reserve` changes the capacity of the `vector` only if the requested space exceeds the current capacity. If the requested size is greater than the current capacity, `reserve` allocates at least as much as (and may allocate more than) the requested amount.

If the requested size is less than or equal to the existing capacity, `reserve` does nothing. In particular, calling `reserve` with a size smaller than `capacity` does not cause the container to give back memory. Thus, after calling `reserve`, the `capacity` will be greater than or equal to the argument passed to `reserve`.

As a result, a call to `reserve` will never reduce the amount of space that the container uses. Similarly, the `resize` members (§ 9.3.5, p. 352) change only the number of elements in the container, not its capacity. We cannot use `resize` to reduce the memory a container holds in reserve.

Under the new library, we can call `shrink_to_fit` to ask a `deque`, `vector`, or `string` to return unneeded memory. This function indicates that we no longer need any excess capacity. However, the implementation is free to ignore this request. There is no guarantee that a call to `shrink_to_fit` will return memory.

capacity and size

It is important to understand the difference between `capacity` and `size`. The `size` of a container is the number of elements it already holds; its `capacity` is how many elements it can hold before more space must be allocated.

The following code illustrates the interaction between `size` and `capacity`:

Click here to view code image

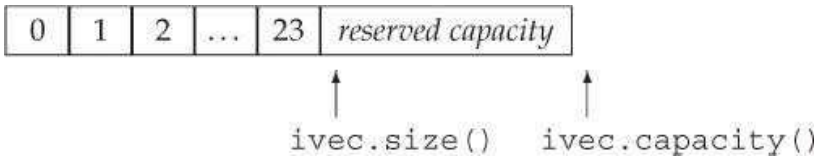
```
vector<int> ivec;
// size should be zero; capacity is implementation defined
cout << "ivec: size: " << ivec.size()
     << " capacity: "  << ivec.capacity() << endl;
// give ivec 24 elements
for (vector<int>::size_type ix = 0; ix != 24; ++ix)
     ivec.push_back(ix);

// size should be 24; capacity will be >= 24 and is implementation defined
cout << "ivec: size: " << ivec.size()
     << " capacity: "  << ivec.capacity() << endl;
```

When run on our system, this code produces the following output:

ivec: size: 0 capacity: 0
ivec: size: 24 capacity: 32

We know that the `size` of an empty `vector` is zero, and evidently our library also sets the `capacity` of an empty `vector` to zero. When we add elements to the `vector`, we know that the `size` is the same as the number of elements we've added. The `capacity` must be at least as large as `size` but can be larger. The details of how much excess capacity is allocated vary by implementations of the library. Under this implementation, adding 24 elements one at a time results in a `capacity` of 32.

Visually we can think of the current state of `ivec` as



We can now `reserve` some additional space:

Click here to view code image

```
ivec.reserve(50); // sets capacity to at least 50; might be more
// size should be 24; capacity will be >= 50 and is implementation defined
cout << "ivec: size: " << ivec.size()
     << " capacity: "  << ivec.capacity() << endl;
```

Here, the output indicates that the call to `reserve` allocated exactly as much space as we requested:

ivec: size: 24 capacity: 50

We might next use up that reserved capacity as follows:

Click here to view code image

```
// add elements to use up the excess capacity
while (ivec.size() != ivec.capacity())
     ivec.push_back(0);
// capacity should be unchanged and size and capacity are now equal
cout << "ivec: size: " << ivec.size()
     << " capacity: "  << ivec.capacity() << endl;
```

The output indicates that at this point we've used up the reserved capacity, and `size` and `capacity` are equal:

ivec: size: 50 capacity: 50

Because we used only reserved capacity, there is no need for the `vector` to do any allocation. In fact, as long as no operation exceeds the `vector`'s capacity, the `vector` must not reallocate its elements.

If we now add another element, the `vector` will have to reallocate itself:

Click here to view code image

```
ivec.push_back(42); // add one more element
// size should be 51; capacity will be >= 51 and is implementation defined
cout << "ivec: size: " << ivec.size()
     << " capacity: "  << ivec.capacity() << endl;
```

The output from this portion of the program

```
ivec: size: 51 capacity: 100
```

indicates that this `vector` implementation appears to follow a strategy of doubling the current capacity each time it has to allocate new storage.

We can call `shrink_to_fit` to ask that memory beyond what is needed for the current size be returned to the system:

Click here to view code image

```
ivec.shrink_to_fit(); // ask for the memory to be returned
// size should be unchanged; capacity is implementation defined
cout << "ivec: size: " << ivec.size()
     << " capacity: "  << ivec.capacity() << endl;
```

Calling `shrink_to_fit` is only a request; there is no guarantee that the library will return the memory.

Note

Each `vector` implementation can choose its own allocation strategy. However, it must not allocate new memory until it is forced to do so.

A `vector` may be reallocated only when the user performs an insert operation when the `size` equals `capacity` or by a call to `resize` or `reserve` with a value that exceeds the current `capacity`. How much memory is allocated beyond the specified amount is up to the implementation.

Every implementation is required to follow a strategy that ensures that it is efficient to use `push_back` to add elements to a `vector`. Technically speaking, the execution time of creating an *n*-element `vector` by calling `push_back` *n* times on an initially empty `vector` must never be more than a constant multiple of *n*.

Exercises Section 9.4

Exercise 9.35: Explain the difference between a `vector`'s `capacity` and its `size`.

Exercise 9.36: Can a container have a `capacity` less than its `size`?

Exercise 9.37: Why don't `list` or `array` have a `capacity` member?

Exercise 9.38: Write a program to explore how `vectors` grow in the library you use.

Exercise 9.39: Explain what the following program fragment does:

Click here to view code image

```
vector<string> svec;
svec.reserve(1024);
string word;
while (cin >> word)
        svec.push_back(word);
svec.resize(svec.size()+svec.size()/2);
```

Exercise 9.40: If the program in the previous exercise reads 256 words, what is its likely `capacity` after it is `resized`? What if it reads 512? 1,000? 1,048?

9.5. Additional `string` Operations

The `string` type provides a number of additional operations beyond those common to the sequential containers. For the most part, these additional operations either support the close interaction between the `string` class and C-style character arrays, or they add versions that let us use indices in place of iterators.

The `string` library defines a great number of functions. Fortunately, these functions use repeated patterns. Given the number of functions supported, this section can be mind-numbing on first reading; so readers might want to skim it. Once you know what kinds of operations are available, you can return for the details when you need to use a particular operation.

9.5.1. Other Ways to Construct `string`s

In addition to the constructors we covered in § 3.2.1 (p. 84) and to the constructors that `string` shares with the other sequential containers (Tables 9.3 (p. 335)) the `string` type supports three more constructors that are described in Table 9.11.

Table 9.11. Additional Ways to Construct `string`s

n, len2 and pos2 are all unsigned values	
`string s(cp, n);`	s is a copy of the first n characters in the array to which cp points. That array must have at least n characters.
`string s(s2, pos2);`	s is a copy of the characters in the `string` s2 starting at the index pos2. Undefined if `pos2 > s2.size()`.
`string s(s2, pos2, len2);`	s is a copy of `len2` characters from s2 starting at the index pos2. Undefined if `pos2 > s2.size()`. Regardless of the value of `len2`, copies at most `s2.size() - pos2` characters.

The constructors that take a `string` or a `const char*` take additional (optional) arguments that let us specify how many characters to copy. When we pass a `string`, we can also specify the index of where to start the copy:

Click here to view code image

```
const char *cp = "Hello World!!!"; // null-terminated array
char noNull[] = {'H', 'i'};          // not null terminated
string s1(cp);  // copy up to the null in cp; s1 == "Hello World!!!"
string s2(noNull,2); // copy two characters from no_null; s2 == "Hi"
string s3(noNull);   // undefined: noNull not null terminated
string s4(cp + 6, 5);// copy 5 characters starting at cp[6]; s4 == "World"
string s5(s1, 6, 5); // copy 5 characters starting at s1[6]; s5 == "World"
string s6(s1, 6);    // copy from s1 [6] to end of s1; s6 == "World!!!"
string s7(s1,6,20);  // ok, copies only to end of s1; s7 == "World!!!"
string s8(s1, 16);   // throws an out_of_range exception
```

Ordinarily when we create a `string` from a `const char*`, the array to which the pointer points must be null terminated; characters are copied up to the null. If we also pass a count, the array does not have to be null terminated. If we do not pass a count and there is no null, or if the given count is greater than the size of the array, the operation is undefined.

When we copy from a `string`, we can supply an optional starting position and a count. The starting position must be less than or equal to the size of the given `string`. If the position is greater than the size, then the constructor throws an `out_of_range` exception (§ 5.6, p. 193). When we pass a count, that many characters are copied, starting from the given position. Regardless of how many characters we ask for, the library copies up to the size of the `string`, but not more.

The substr Operation

The `substr` operation (described in Table 9.12) returns a `string` that is a copy of part or all of the original `string`. We can pass `substr` an optional starting position and count:

Click here to view code image

```
string s("hello world");
string s2 = s.substr(0, 5);   // s2 = hello
string s3 = s.substr(6);      // s3 = world
string s4 = s.substr(6, 11); // s3 = world
string s5 = s.substr(12);     // throws an out_of_range exception
```

Table 9.12. Substring Operation

`s.substr(pos, n)`	Return a `string` containing `n` characters from `s` starting at `pos`. `pos` defaults to 0. `n` defaults to a value that causes the library to copy all the characters in `s` starting from `pos`.

The `substr` function throws an `out_of_range` exception (§ 5.6, p. 193) if the position exceeds the size of the `string`. If the position plus the count is greater than the size, the count is adjusted to copy only up to the end of the `string`.

Exercises Section 9.5.1

Exercise 9.41: Write a program that initializes a `string` from a `vector<char>`.

Exercise 9.42: Given that you want to read a character at a time into a `string`, and you know that you need to read at least 100 characters, how might you improve the performance of your program?

9.5.2. Other Ways to Change a `string`

The `string` type supports the sequential container assignment operators and the `assign`, `insert`, and `erase` operations (§ 9.2.5, p. 337, § 9.3.1, p. 342, and § 9.3.3, p. 348). It also defines additional versions of `insert` and `erase`.

In addition to the versions of `insert` and `erase` that take iterators, `string` provides versions that take an index. The index indicates the starting element to `erase` or the position before which to `insert` the given values:

Click here to view code image

```
s.insert(s.size(), 5, '!'); // insert five exclamation points at the end of s
s.erase(s.size() - 5, 5);   // erase the last five characters from s
```

The `string` library also provides versions of `insert` and `assign` that take C-style character arrays. For example, we can use a null-terminated character array as the

value to `insert` or `assign` into a `string`:

Click here to view code image

```
const char *cp = "Stately, plump Buck";
s.assign(cp, 7);             //   s == "Stately"
s.insert(s.size(), cp + 7); //   s == "Stately, plump Buck"
```

Here we first replace the contents of `s` by calling `assign`. The characters we assign into `s` are the seven characters starting with the one pointed to by `cp`. The number of characters we request must be less than or equal to the number of characters (excluding the null terminator) in the array to which `cp` points.

When we call `insert` on `s`, we say that we want to insert the characters before the (nonexistent) element at `s[size()]`. In this case, we copy characters starting seven characters past `cp` up to the terminating null.

We can also specify the characters to `insert` or `assign` as coming from another `string` or substring thereof:

Click here to view code image

```
string s = "some string", s2 = "some other string";
s.insert(0, s2); // insert a copy of s2 before position 0 in s
// insert s2.size() characters from s2 starting at s2[0] before s[0]
s.insert(0, s2, 0, s2.size());
```

The append and replace Functions

The `string` class defines two additional members, `append` and `replace`, that can change the contents of a `string`. Table 9.13 summarizes these functions. The `append` operation is a shorthand way of inserting at the end:

Click here to view code image

```
string s("C++ Primer"), s2 = s; // initialize s and s2 to "C++ Primer"
s.insert(s.size(), " 4th Ed."); // s == "C++ Primer 4th Ed."
s2.append(" 4th Ed."); // equivalent: appends " 4th Ed." to s2; s == s2
```

Table 9.13. Operations to Modify strings

`s.insert(pos, args)`	Insert characters specified by *args* before *pos*. *pos* can be an index or an iterator. Versions taking an index return a reference to `s`; those taking an iterator return an iterator denoting the first inserted character.
`s.erase(pos, len)`	Remove `len` characters starting at position `pos`. If `len` is omitted, removes characters from `pos` to the end of the `s`. Returns a reference to `s`.
`s.assign(args)`	Replace characters in `s` according to *args*. Returns a reference to `s`.
`s.append(args)`	Append *args* to `s`. Returns a reference to `s`.
`s.replace(range, args)`	Remove *range* of characters from `s` and replace them with the characters formed by *args*. *range* is either an index and a length or a pair of iterators into `s`. Returns a reference to `s`.

***args* can be one of the following; `append` and `assign` can use all forms**
`str` must be distinct from `s` and the iterators `b` and `e` may not refer to `s`

`str`	The `string` `str`.
`str, pos, len`	Up to `len` characters from `str` starting at `pos`.
`cp, len`	Up to `len` characters from the character array pointed to by `cp`.
`cp`	Null-terminated array pointed to by pointer `cp`.
`n, c`	`n` copies of character `c`.
`b, e`	Characters in the range formed by iterators `b` and `e`.
initializer list	Comma-separated list of characters enclosed in braces.

***args* for `replace` and `insert` depend on how *range* or *pos* is specified.**

`replace` `(pos,len,args)`	`replace` `(b,e,args)`	`insert` `(pos,args)`	`insert` `(iter,args)`	*args* can be
yes	yes	yes	no	`str`
yes	no	yes	no	`str, pos, len`
yes	yes	yes	no	`cp, len`
yes	yes	no	no	`cp`
yes	yes	yes	yes	`n, c`
no	yes	no	yes	`b2, e2`
no	yes	no	yes	*initializer list*

The `replace` operations are a shorthand way of calling `erase` and `insert`:

Click here to view code image

```
// equivalent way to replace "4th" by "5th"
s.erase(11, 3);                    // s == "C++ Primer Ed."
s.insert(11, "5th");               // s == "C++ Primer 5th Ed."
// starting at position 11, erase three characters and then insert "5th"
s2.replace(11, 3, "5th"); // equivalent: s == s2
```

In the call to `replace`, the text we inserted happens to be the same size as the text we removed. We can insert a larger or smaller `string`:

Click here to view code image

```
s.replace(11, 3, "Fifth");       // s == "C++ Primer Fifth Ed."
```

In this call we remove three characters but insert five in their place.

The Many Overloaded Ways to Change a string

The `append`, `assign`, `insert`, and `replace` functions listed Table 9.13 have several overloaded versions. The arguments to these functions vary as to how we specify what characters to add and what part of the `string` to change. Fortunately, these functions share a common interface.

The `assign` and `append` functions have no need to specify what part of the `string` is changed: `assign` always replaces the entire contents of the `string` and `append` always adds to the end of the `string`.

The `replace` functions provide two ways to specify the range of characters to remove. We can specify that range by a position and a length, or with an iterator range. The `insert` functions give us two ways to specify the insertion point: with either an index or an iterator. In each case, the new element(s) are inserted in front of the given index or iterator.

There are several ways to specify the characters to add to the `string`. The new characters can be taken from another `string`, from a character pointer, from a brace-enclosed list of characters, or as a character and a count. When the characters come from a `string` or a character pointer, we can pass additional arguments to control whether we copy some or all of the characters from the argument.

Not every function supports every version of these arguments. For example, there is no version of `insert` that takes an index and an initializer list. Similarly, if we want to specify the insertion point using an iterator, then we cannot pass a character pointer as the source for the new characters.

Exercises Section 9.5.2

Exercise 9.43: Write a function that takes three `strings`, `s`, `oldVal`, and `newVal`. Using iterators, and the `insert` and `erase` functions replace all instances of `oldVal` that appear in `s` by `newVal`. Test your function by using it to replace common abbreviations, such as “tho” by “though” and “thru” by “through”.

Exercise 9.44: Rewrite the previous function using an index and `replace`.

Exercise 9.45: Write a funtion that takes a `string` representing a name and two other `strings` representing a prefix, such as “Mr.” or “Ms.” and a suffix, such as “Jr.” or “III”. Using iterators and the `insert` and `append` functions, generate and return a new `string` with the suffix and prefix added to the given name.

Exercise 9.46: Rewrite the previous exercise using a position and length to manage the `strings`. This time use only the `insert` function.

9.5.3. string Search Operations

The `string` class provides six different search functions, each of which has four overloaded versions. Table 9.14 describes the search members and their arguments. Each of these search operations returns a `string::size_type` value that is the index of where the match occurred. If there is no match, the function returns a `static` member (§ 7.6, p. 300) named `string::npos`. The library defines `npos` as a `const string::size_type` initialized with the value `-1`. Because `npos` is an `unsigned` type, this initializer means `npos` is equal to the largest possible size any `string` could have (§ 2.1.2, p. 35).

Table 9.14. string Search Operations

Search operations return the index of the desired character or npos if not found	
`s.find(args)`	Find the first occurrence of *args* in `s`.
`s.rfind(args)`	Find the last occurrence of *args* in `s`.
`s.find_first_of(args)`	Find the first occurrence of any character from *args* in `s`.
`s.find_last_of(args)`	Find the last occurrence of any character from *args* in `s`.
`s.find_first_not_of(args)`	Find the first character in `s` that is not in *args*.
`s.find_last_not_of(args)`	Find the last character in `s` that is not in *args*.
***args* must be one of**	
`c, pos`	Look for the character `c` starting at position `pos` in `s`. `pos` defaults to 0.
`s2, pos`	Look for the `string` `s2` starting at position `pos` in `s`. `pos` defaults to 0.
`cp, pos`	Look for the C-style null-terminated string pointed to by the pointer `cp`. Start looking at position `pos` in `s`. `pos` defaults to 0.
`cp, pos, n`	Look for the first `n` characters in the array pointed to by the pointer `cp`. Start looking at position `pos` in `s`. No default for `pos` or `n`.

Warning

The `string` search functions return `string::size_type`, which is an `unsigned` type. As a result, it is a bad idea to use an `int`, or other signed type, to hold the return from these functions (§ 2.1.2, p. 36).

The `find` function does the simplest search. It looks for its argument and returns the index of the first match that is found, or `npos` if there is no match:

Click here to view code image

```
string name("AnnaBelle");
```

```
auto pos1 = name.find("Anna"); // pos1 == 0
```

returns `0`, the index at which the substring `"Anna"` is found in `"AnnaBelle"`.

Searching (and other `string` operations) are case sensitive. When we look for a value in the `string`, case matters:

Click here to view code image

```
string lowercase("annabelle");
pos1 = lowercase.find("Anna");   // pos1 == npos
```

This code will set `pos1` to `npos` because `Anna` does not match `anna`.

A slightly more complicated problem requires finding a match to any character in the search string. For example, the following locates the first digit within `name`:

Click here to view code image

```
string numbers("0123456789"), name("r2d2");
// returns 1, i.e., the index of the first digit in name
auto pos = name.find_first_of(numbers);
```

Instead of looking for a match, we might call `find_first_not_of` to find the first position that is *not* in the search argument. For example, to find the first nonnumeric character of a `string`, we can write

Click here to view code image

```
string dept("03714p3");
// returns 5, which is the index to the character 'p'
auto pos = dept.find_first_not_of(numbers);
```

Specifying Where to Start the Search

We can pass an optional starting position to the `find` operations. This optional argument indicates the position from which to start the search. By default, that position is set to zero. One common programming pattern uses this optional argument to loop through a `string` finding all occurrences:

Click here to view code image

```
string::size_type pos = 0;
// each iteration finds the next number in name
while ((pos = name.find_first_of(numbers, pos))
               != string::npos) {
    cout << "found number at index: " << pos
         << " element is " << name[pos] << endl;
    ++pos; // move to the next character
}
```

The condition in the `while` resets `pos` to the index of the first number encountered, starting from the current value of `pos`. So long as `find_first_of` returns a valid

index, we print the current result and increment `pos`.

Had we neglected to increment `pos`, the loop would never terminate. To see why, consider what would happen if we didn't do the increment. On the second trip through the loop we start looking at the character indexed by `pos`. That character would be a number, so `find_first_of` would (repeatedly) returns `pos`!

Searching Backward

The `find` operations we've used so far execute left to right. The library provides analogous operations that search from right to left. The `rfind` member searches for the last—that is, right-most—occurrence of the indicated substring:

Click here to view code image

```
string river("Mississippi");
auto first_pos = river.find("is");   // returns 1
auto last_pos = river.rfind("is");   // returns 4
```

`find` returns an index of 1, indicating the start of the first `"is"`, while `rfind` returns an index of 4, indicating the start of the last occurrence of `"is"`.

Similarly, the `find_last` functions behave like the `find_first` functions, except that they return the *last* match rather than the first:

- `find_last_of` searches for the last character that matches any element of the search `string`.
- `find_last_not_of` searches for the last character that does not match any element of the search `string`.

Each of these operations takes an optional second argument indicating the position within the `string` to begin searching.

9.5.4. The compare Functions

In addition to the relational operators (§ 3.2.2, p. 88), the `string` library provides a set of `compare` functions that are similar to the C library `strcmp` function (§ 3.5.4, p. 122). Like `strcmp`, `s.compare` returns zero or a positive or negative value depending on whether `s` is equal to, greater than, or less than the string formed from the given arguments.

Exercises Section 9.5.3

Exercise 9.47: Write a program that finds each numeric character and then each alphabetic character in the `string` `"ab2c3d7R4E6"`. Write two versions of the program. The first should use `find_first_of`, and the

second `find_first_not_of`.

Exercise 9.48: Given the definitions of `name` and `numbers` on page 365, what does `numbers.find(name)` return?

Exercise 9.49: A letter has an ascender if, as with `d` or `f`, part of the letter extends above the middle of the line. A letter has a descender if, as with `p` or `g`, part of the letter extends below the line. Write a program that reads a file containing words and reports the longest word that contains neither ascenders nor descenders.

As shown in Table 9.15, there are six versions of `compare`. The arguments vary based on whether we are comparing two `strings` or a `string` and a character array. In both cases, we might compare the entire string or a portion thereof.

Table 9.15. Possible Arguments to s.compare

`s2`	Compare `s` to `s2`.
`pos1, n1, s2`	Compares `n1` characters starting at `pos1` from `s` to `s2`.
`pos1, n1, s2, pos2, n2`	Compares `n1` characters starting at `pos1` from `s` to the `n2` characters starting at `pos2` in `s2`.
`cp`	Compares `s` to the null-terminated array pointed to by `cp`.
`pos1, n1, cp`	Compares `n1` characters starting at `pos1` from `s` to `cp`.
`pos1, n1, cp, n2`	Compares `n1` characters starting at `pos1` from `s` to `n2` characters starting from the pointer `cp`.

9.5.5. Numeric Conversions

Strings often contain characters that represent numbers. For example, we represent the numeric value 15 as a `string` with two characters, the character `'1'` followed by the character `'5'`. In general, the character representation of a number differs from its numeric value. The numeric value 15 stored in a 16-bit `short` has the bit pattern `0000000000001111`, whereas the character string `"15"` represented as two Latin-1 `chars` has the bit pattern `0011000100110101`. The first byte represents the character `'1'` which has the octal value 061, and the second byte represents `'5'`, which in Latin-1 is octal 065.

The new standard introduced several functions that convert between numeric data and library `strings`:

Click here to view code image

```
int i = 42;
string s = to_string(i);    // converts the int i to its character
```

representation

```
double d = stod(s);          // converts the string s to floating-point
```

Table 9.16. Conversions between strings and Numbers

`to_string(val);`	Overloaded functions returning the `string` representation of `val`. `val` can be any arithmetic type (§ 2.1.1, p. 32). There are versions of `to_string` for each floating-point type and integral type that is `int` or larger. Small integral types are promoted (§ 4.11.1, p. 160) as usual.

`stoi(s, p, b)`
`stol(s, p, b)`
`stoul(s, p, b)`
`stoll(s, p, b)`
`stoull(s, p, b)`
	Return the initial substring of `s` that has numeric content as an `int`, `long`, `unsigned long`, `long long`, `unsigned long long`, respectively. `b` indicates the numeric base to use for the conversion; `b` defaults to 10. `p` is a pointer to a `size_t` in which to put the index of the first nonnumeric character in `s`; `p` defaults to 0, in which case the function does not store the index.

`stof(s, p)`
`stod(s, p)`
`stold(s, p)`
	Return the initial numeric substring in `s` as a `float`, `double`, or `long double`, respectively. `p` has the same behavior as described for the integer conversions.

Here we call `to_string` to convert `42` to its corresponding `string` representation and then call `stod` to convert that `string` to floating-point.

The first non-whitespace character in the `string` we convert to numeric value must be a character that can appear in a number:

Click here to view code image

```
string s2 = "pi = 3.14";
// convert the first substring in s that starts with a digit, d = 3.14
d = stod(s2.substr(s2.find_first_of("+-.0123456789")));
```

In this call to `stod`, we call `find_first_of` (§ 9.5.3, p. 364) to get the position of the first character in `s` that could be part of a number. We pass the substring of `s` starting at that position to `stod`. The `stod` function reads the `string` it is given until it finds a character that cannot be part of a number. It then converts the character representation of the number it found into the corresponding double-precision floating-point value.

The first non-whitespace character in the `string` must be a sign (`+` or `-`) or a digit. The `string` can begin with `0x` or `0X` to indicate hexadecimal. For the functions that convert to floating-point the `string` may also start with a decimal point (`.`) and may contain an `e` or `E` to designate the exponent. For the functions that convert to integral type, depending on the base, the `string` can contain alphabetic characters corresponding to numbers beyond the digit 9.

Note

If the `string` can't be converted to a number, These functions throw an `invalid_argument` exception (§ 5.6, p. 193). If the conversion generates a

value that can't be represented, they throw `out_of_range`.

9.6. Container Adaptors

In addition to the sequential containers, the library defines three sequential container adaptors: `stack`, `queue`, and `priority_queue`. An **adaptor** is a general concept in the library. There are container, iterator, and function adaptors. Essentially, an adaptor is a mechanism for making one thing act like another. A container adaptor takes an existing container type and makes it act like a different type. For example, the `stack` adaptor takes a sequential container (other than `array` or `forward_list`) and makes it operate as if it were a `stack`. Table 9.17 lists the operations and types that are common to all the container adaptors.

Table 9.17. Operations and Types Common to the Container Adaptors

`size_type`	Type large enough to hold the size of the largest object of this type.
`value_type`	Element type.
`container_type`	Type of the underlying container on which the adaptor is implemented.
`A a;`	Create a new empty adaptor named `a`.
`A a(c);`	Create a new adaptor named `a` with a copy of the container `c`.
relational operators	Each adaptor supports all the relational operators: `==`, `!=`, `<`, `<=`, `>`, `>=`. These operators return the result of comparing the underlying containers.
`a.empty()`	`false` if `a` has any elements, `true` otherwise.
`a.size()`	Number of elements in `a`.
`swap(a, b)` `a.swap(b)`	Swaps the contents of `a` and `b`; `a` and `b` must have the same type, including the type of the container on which they are implemented.

Exercises Section 9.5.5

Exercise 9.50: Write a program to process a `vector<string>`s whose elements represent integral values. Produce the sum of all the elements in that `vector`. Change the program so that it sums of `strings` that represent floating-point values.

Exercise 9.51: Write a class that has three `unsigned` members representing year, month, and day. Write a constructor that takes a `string` representing a date. Your constructor should handle a variety of date formats, such as January 1, 1900, 1/1/1900, Jan 1, 1900, and so on.

Defining an Adaptor

Each adaptor defines two constructors: the default constructor that creates an empty object, and a constructor that takes a container and initializes the adaptor by copying the given container. For example, assuming that `deq` is a `deque<int>`, we can use `deq` to initialize a new `stack` as follows:

Click here to view code image

```
stack<int> stk(deq);   // copies elements from deq into stk
```

By default both `stack` and `queue` are implemented in terms of `deque`, and a `priority_queue` is implemented on a `vector`. We can override the default container type by naming a sequential container as a second type argument when we create the adaptor:

Click here to view code image

```
// empty stack implemented on top of vector
stack<string, vector<string>> str_stk;
// str_stk2 is implemented on top of vector and initially holds a copy of svec
stack<string, vector<string>> str_stk2(svec);
```

There are constraints on which containers can be used for a given adaptor. All of the adaptors require the ability to add and remove elements. As a result, they cannot be built on an `array`. Similarly, we cannot use `forward_list`, because all of the adaptors require operations that add, remove, or access the last element in the container. A `stack` requires only `push_back`, `pop_back`, and `back` operations, so we can use any of the remaining container types for a `stack`. The `queue` adaptor requires `back`, `push_back`, `front`, and `push_front`, so it can be built on a `list` or `deque` but not on a `vector`. A `priority_queue` requires random access in addition to the `front`, `push_back`, and `pop_back` operations; it can be built on a `vector` or a `deque` but not on a `list`.

Stack Adaptor

The `stack` type is defined in the `stack` header. The operations provided by a `stack` are listed in Table 9.18. The following program illustrates the use of `stack`:

Click here to view code image

```
stack<int> intStack;   // empty stack
// fill up the stack
for (size_t ix = 0; ix != 10; ++ix)
    intStack.push(ix);    // intStackholds 0 ... 9 inclusive
while (!intStack.empty()) {     // while there are still values in intStack
    int value = intStack.top();
    // code that uses value
```

```
        intStack.pop(); // pop the top element, and repeat
    }
```

Table 9.18. Stack Operations in Addition to Those in Table 9.17

Uses `deque` by default; can be implemented on a `list` or `vector` as well.	
`s.pop()`	Removes, but does not return, the top element from the `stack`.
`s.push(item)`	Creates a new top element on the `stack` by copying or moving `item`, or
`s.emplace(args)`	by constructing the element from *args*.
`s.top()`	Returns, but does not remove, the top element on the `stack`.

The declaration

Click here to view code image

```
stack<int> intStack;  // empty stack
```

defines `intStack` to be an empty `stack` that holds integer elements. The `for` loop adds ten elements, initializing each to the next integer in sequence starting from zero. The `while` loop iterates through the entire `stack`, examining the `top` value and popping it from the `stack` until the `stack` is empty.

Each container adaptor defines its own operations in terms of operations provided by the underlying container type. We can use only the adaptor operations and cannot use the operations of the underlying container type. For example,

Click here to view code image

```
intStack.push(ix);    // intStackholds 0 ... 9 inclusive
```

calls `push_back` on the `deque` object on which `intStack` is based. Although `stack` is implemented by using a `deque`, we have no direct access to the `deque` operations. We cannot call `push_back` on a `stack`; instead, we must use the `stack` operation named `push`.

The Queue Adaptors

The `queue` and `priority_queue` adaptors are defined in the `queue` header. Table 9.19 lists the operations supported by these types.

Table 9.19. `queue`, `priority_queue` Operations in Addition to Table 9.17

By default queue uses deque and priority_queue uses vector; queue can use a list or vector as well, priority_queue can use a deque.	
`q.pop()`	Removes, but does not return, the front element or highest-priority element from the queue or `priority_queue`, respectively.
`q.front()`	Returns, but does not remove, the front or back element of `q`.
`q.back()`	**Valid only for queue**
`q.top()`	Returns, but does not remove, the highest-priority element. **Valid only for `priority_queue`.**
`q.push(item)` `q.emplace(args)`	Create an element with value `item` or constructed from *args* at the end of the queue or in its appropriate position in `priority_queue`.

The library `queue` uses a first-in, first-out (FIFO) storage and retrieval policy. Objects entering the queue are placed in the back and objects leaving the queue are removed from the front. A restaurant that seats people in the order in which they arrive is an example of a FIFO queue.

A `priority_queue` lets us establish a priority among the elements held in the queue. Newly added elements are placed ahead of all the elements with a lower priority. A restaurant that seats people according to their reservation time, regardless of when they arrive, is an example of a priority queue. By default, the library uses the < operator on the element type to determine relative priorities. We'll learn how to override this default in § 11.2.2 (p. 425).

Exercises Section 9.6

Exercise 9.52: Use a `stack` to process parenthesized expressions. When you see an open parenthesis, note that it was seen. When you see a close parenthesis after an open parenthesis, `pop` elements down to and including the open parenthesis off the `stack`. `push` a value onto the `stack` to indicate that a parenthesized expression was replaced.

Chapter Summary

The library containers are template types that holds objects of a given type. In a sequential container, elements are ordered and accessed by position. The sequential containers share a common, standardized interface: If two sequential containers offer a particular operation, then the operation has the same interface and meaning for both containers.

All the containers (except `array`) provide efficient dynamic memory management. We may add elements to the container without worrying about where to store the elements. The container itself manages its storage. Both `vector` and `string` provide more detailed control over memory management through their `reserve` and `capacity` members.

For the most part, the containers define surprisingly few operations. Containers define constructors, operations to add or remove elements, operations to determine the size of the container, and operations to return iterators to particular elements. Other useful operations, such as sorting or searching, are defined not by the container types but by the standard algorithms, which we shall cover in Chapter 10.

When we use container operations that add or remove elements, it is essential to remember that these operations can invalidate iterators, pointers, or references to elements in the container. Many operations that invalidate an iterator, such as `insert` or `erase`, return a new iterator that allows the programmer to maintain a position within the container. Loops that use container operations that change the size of a container should be particularly careful in their use of iterators, pointers, and references.

Defined Terms

adaptor Library type, function, or iterator that, given a type, function, or iterator, makes it act like another. There are three sequential container adaptors: `stack`, `queue`, and `priority_queue`. Each adaptor defines a new interface on top of an underlying sequential container type.

array Fixed-size sequential container. To define an `array`, we must give the size in addition to specifying the element type. Elements in an `array` can be accessed by their positional index. Supports fast random access to elements.

begin Container operation that returns an iterator referring to the first element in the container, if there is one, or the off-the-end iterator if the container is empty. Whether the returned iterator is `const` depends on the type of the container.

cbegin Container operation that returns a `const_iterator` referring to the first element in the container, if there is one, or the off-the-end iterator if the container is empty.

cend Container operation that returns a `const_iterator` referring to the (nonexistent) element one past the end of the container.

container Type that holds a collection of objects of a given type. Each library container type is a template type. To define a container, we must specify the type of the elements stored in the container. With the exception of `array`, the library containers are variable-size.

deque Sequential container. Elements in a `deque` can be accessed by their positional index. Supports fast random access to elements. Like a `vector` in all respects except that it supports fast insertion and deletion at the front of the container as well as at the back and does not relocate elements as a result of insertions or deletions at either end.

end Container operation that returns an iterator referring to the (nonexistent) element one past the end of the container. Whether the returned iterator is `const` depends on the type of the container.

forward_list Sequential container that represents a singly linked list. Elements in a `forward_list` may be accessed only sequentially; starting from a given element, we can get to another element only by traversing each element between them. Iterators on `forward_list` do not support decrement (`--`). Supports fast insertion (or deletion) anywhere in the `forward_list`. Unlike other containers, insertions and deletions occur *after* a given iterator position. As a consequence, `forward_list` has a "before-the-beginning" iterator to go along with the usual off-the-end iterator. Iterators remain valid when new elements are added. When an element is removed, only the iterators to that element are invalidated.

iterator range Range of elements denoted by a pair of iterators. The first iterator denotes the first element in the sequence, and the second iterator denotes one past the last element. If the range is empty, then the iterators are equal (and vice versa—if the iterators are unequal, they denote a nonempty range). If the range is not empty, then it must be possible to reach the second iterator by repeatedly incrementing the first iterator. By incrementing the iterator, each element in the sequence can be processed.

left-inclusive interval A range of values that includes its first element but not its last. Typically denoted as `[i, j)`, meaning the sequence starting at and including `i` up to but excluding `j`.

list Sequential container representing a doubly linked list. Elements in a `list` may be accessed only sequentially; starting from a given element, we can get to another element only by traversing each element between them. Iterators on `list` support both increment (`++`) and decrement (`--`). Supports fast insertion (or deletion) anywhere in the `list`. Iterators remain valid when new elements are added. When an element is removed, only the iterators to that element are invalidated.

off-the-beginning iterator Iterator denoting the (nonexistent) element just before the beginning of a `forward_list`. Returned from the `forward_list` member `before_begin`. Like the `end()` iterator, it may not be dereferenced.

off-the-end iterator Iterator that denotes one past the last element in the range. Commonly referred to as the "end iterator".

priority_queue Adaptor for the sequential containers that yields a queue in which elements are inserted, not at the end but according to a specified priority level. By default, priority is determined by using the less-than operator for the element type.

queue Adaptor for the sequential containers that yields a type that lets us add elements to the back and remove elements from the front.

sequential container Type that holds an ordered collection of objects of a single type. Elements in a sequential container are accessed by position.

stack Adaptor for the sequential containers that yields a type that lets us add and remove elements from one end only.

vector Sequential container. Elements in a `vector` can be accessed by their positional index. Supports fast random access to elements. We can efficiently add or remove `vector` elements only at the back. Adding elements to a `vector` might cause it to be reallocated, invalidating all iterators into the `vector`. Adding (or removing) an element in the middle of a `vector` invalidates all iterators to elements after the insertion (or deletion) point.

Chapter 10. Generic Algorithms

Contents

The library containers define a surprisingly small set of operations. Rather than adding lots of functionality to each container, the library provides a set of algorithms, most of which are independent of any particular container type. These algorithms are *generic*: They operate on different types of containers and on elements of various types.

The generic algorithms, and a more detailed look at iterators, form the subject matter of this chapter.

The sequential containers define few operations: For the most part, we can add and remove elements, access the first or last element, determine whether a container is empty, and obtain iterators to the first or one past the last element.

We can imagine many other useful operations one might want to do: We might want to find a particular element, replace or remove a particular value, reorder the container elements, and so on.

Rather than define each of these operations as members of each container type, the standard library defines a set of **generic algorithms**: “algorithms” because they

implement common classical algorithms such as sorting and searching, and “generic” because they operate on elements of differing type and across multiple container types—not only library types such as `vector` or `list`, but also the built-in array type—and, as we shall see, over other kinds of sequences as well.

10.1. Overview

Most of the algorithms are defined in the `algorithm` header. The library also defines a set of generic numeric algorithms that are defined in the `numeric` header.

In general, the algorithms do not work directly on a container. Instead, they operate by traversing a range of elements bounded by two iterators (§ 9.2.1, p. 331). Typically, as the algorithm traverses the range, it does something with each element. For example, suppose we have a `vector` of `int`s and we want to know if that `vector` holds a particular value. The easiest way to answer this question is to call the library `find` algorithm:

Click here to view code image

```
int val = 42; // value we'll look for
// result will denote the element we want if it's in vec, or vec.cend() if not
auto result = find(vec.cbegin(), vec.cend(), val);
// report the result
cout << "The value " << val
     << (result == vec.cend()
           ? " is not present" : " is present") << endl;
```

The first two arguments to `find` are iterators denoting a range of elements, and the third argument is a value. `find` compares each element in the given range to the given value. It returns an iterator to the first element that is equal to that value. If there is no match, `find` returns its second iterator to indicate failure. Thus, we can determine whether the element was found by comparing the return value with the second iterator argument. We do this test in the output statement, which uses the conditional operator (§ 4.7, p. 151) to report whether the value was found.

Because `find` operates in terms of iterators, we can use the same `find` function to look for values in any type of container. For example, we can use `find` to look for a value in a `list` of `string`s:

Click here to view code image

```
string val = "a value";  // value we'll look for
// this call to find looks through string elements in a list
auto result = find(lst.cbegin(), lst.cend(), val);
```

Similarly, because pointers act like iterators on built-in arrays, we can use `find` to look in an array:

Click here to view code image

```
int ia[] = {27, 210, 12, 47, 109, 83};
int val = 83;
int* result = find(begin(ia), end(ia), val);
```

Here we use the library `begin` and `end` functions (§ 3.5.3, p. 118) to pass a pointer to the first and one past the last elements in `ia`.

We can also look in a subrange of the sequence by passing iterators (or pointers) to the first and one past the last element of that subrange. For example, this call looks for a match in the elements `ia[1]`, `ia[2]`, and `ia[3]`:

Click here to view code image

```
// search the elements starting from ia[1] up to but not including ia[4]
auto result = find(ia + 1, ia + 4, val);
```

How the Algorithms Work

To see how the algorithms can be used on varying types of containers, let's look a bit more closely at `find`. Its job is to find a particular element in an unsorted sequence of elements. Conceptually, we can list the steps `find` must take:

1. It accesses the first element in the sequence.
2. It compares that element to the value we want.
3. If this element matches the one we want, `find` returns a value that identifies this element.
4. Otherwise, `find` advances to the next element and repeats steps 2 and 3.
5. `find` must stop when it has reached the end of the sequence.
6. If `find` gets to the end of the sequence, it needs to return a value indicating that the element was not found. This value and the one returned from step 3 must have compatible types.

None of these operations depends on the type of the container that holds the elements. So long as there is an iterator that can be used to access the elements, `find` doesn't depend in any way on the container type (or even whether the elements are stored in a container).

Iterators Make the Algorithms Container Independent, ...

All but the second step in the `find` function can be handled by iterator operations: The iterator dereference operator gives access to an element's value; if a matching element is found, `find` can return an iterator to that element; the iterator increment operator moves to the next element; the "off-the-end" iterator will indicate when `find` has reached the end of its given sequence; and `find` can return the off-the-end

iterator (§ 9.2.1, p. 331) to indicate that the given value wasn't found.

...But Algorithms Do Depend on Element-Type Operations

Although iterators make the algorithms container independent, most of the algorithms use one (or more) operation(s) on the element type. For example, step 2, uses the element type's == operator to compare each element to the given value.

Other algorithms require that the element type have the < operator. However, as we'll see, most algorithms provide a way for us to supply our own operation to use in place of the default operator.

Exercises Section 10.1

Exercise 10.1: The `algorithm` header defines a function named `count` that, like `find`, takes a pair of iterators and a value. `count` returns a count of how often that value appears. Read a sequence of `ints` into a `vector` and print the `count` of how many elements have a given value.

Exercise 10.2: Repeat the previous program, but read values into a `list` of `strings`.

Key Concept: Algorithms Never Execute Container Operations

The generic algorithms do not themselves execute container operations. They operate solely in terms of iterators and iterator operations. The fact that the algorithms operate in terms of iterators and not container operations has a perhaps surprising but essential implication: Algorithms never change the size of the underlying container. Algorithms may change the values of the elements stored in the container, and they may move elements around within the container. They do not, however, ever add or remove elements directly.

As we'll see in § 10.4.1 (p. 401), there is a special class of iterator, the inserters, that do more than traverse the sequence to which they are bound. When we assign to these iterators, they execute insert operations on the underlying container. When an algorithm operates on one of these iterators, the *iterator* may have the effect of adding elements to the container. The *algorithm* itself, however, never does so.

10.2. A First Look at the Algorithms

The library provides more than 100 algorithms. Fortunately, like the containers, the

algorithms have a consistent architecture. Understanding this architecture makes learning and using the algorithms easier than memorizing all 100+ of them. In this chapter, we'll illustrate how to use the algorithms, and describe the unifying principles that characterize them. Appendix A lists all the algorithms classified by how they operate.

With only a few exceptions, the algorithms operate over a range of elements. We'll refer to this range as the "input range." The algorithms that take an input range always use their first two parameters to denote that range. These parameters are iterators denoting the first and one past the last elements to process.

Although most algorithms are similar in that they operate over an input range, they differ in how they use the elements in that range. The most basic way to understand the algorithms is to know whether they read elements, write elements, or rearrange the order of the elements.

10.2.1. Read-Only Algorithms

A number of the algorithms read, but never write to, the elements in their input range. The `find` function is one such algorithm, as is the `count` function we used in the exercises for § 10.1 (p. 378).

Another read-only algorithm is `accumulate`, which is defined in the `numeric` header. The `accumulate` function takes three arguments. The first two specify a range of elements to sum. The third is an initial value for the sum. Assuming `vec` is a sequence of integers, the following

Click here to view code image

```
// sum the elements in vec starting the summation with the value 0
int sum = accumulate(vec.cbegin(), vec.cend(), 0);
```

sets `sum` equal to the sum of the elements in `vec`, using `0` as the starting point for the summation.

Note

The type of the third argument to `accumulate` determines which addition operator is used and is the type that `accumulate` returns.

Algorithms and Element Types

The fact that `accumulate` uses its third argument as the starting point for the summation has an important implication: It must be possible to add the element type

to the type of the sum. That is, the elements in the sequence must match or be convertible to the type of the third argument. In this example, the elements in `vec` might be `int`s, or they might be `double`, or `long long`, or any other type that can be added to an `int`.

As another example, because `string` has a + operator, we can concatenate the elements of a `vector` of `string`s by calling `accumulate`:

Click here to view code image

```
string sum = accumulate(v.cbegin(), v.cend(), string(""));
```

This call concatenates each element in `v` onto a `string` that starts out as the empty string. Note that we explicitly create a `string` as the third parameter. Passing the empty string as a string literal would be a compile-time error:

Click here to view code image

```
// error: no + on const char*
string sum = accumulate(v.cbegin(), v.cend(), "");
```

Had we passed a string literal, the type of the object used to hold the sum would be `const char*`. That type determines which + operator is used. Because there is no + operator for type `const char*`, this call will not compile.

Best Practices

Ordinarily it is best to use `cbegin()` and `cend()` (§ 9.2.3, p. 334) with algorithms that read, but do not write, the elements. However, if you plan to use the iterator returned by the algorithm to change an element's value, then you need to pass `begin()` and `end()`.

Algorithms That Operate on Two Sequences

Another read-only algorithm is `equal`, which lets us determine whether two sequences hold the same values. It compares each element from the first sequence to the corresponding element in the second. It returns `true` if the corresponding elements are equal, `false` otherwise. The algorithm takes three iterators: The first two (as usual) denote the range of elements in the first sequence; the third denotes the first element in the second sequence:

Click here to view code image

```
// roster2 should have at least as many elements as roster1
equal(roster1.cbegin(), roster1.cend(), roster2.cbegin());
```

Because `equal` operates in terms of iterators, we can call `equal` to compare

elements in containers of different types. Moreover, the element types also need not be the same so long as we can use == to compare the element types. For example, `roster1` could be a `vector<string>` and `roster2` a `list<const char*>`.

However, `equal` makes one critically important assumption: It assumes that the second sequence is at least as big as the first. This algorithm potentially looks at every element in the first sequence. It assumes that there is a corresponding element for each of those elements in the second sequence.

Warning

Algorithms that take a single iterator denoting a second sequence *assume* that the second sequence is at least as large at the first.

Exercises Section 10.2.1

Exercise 10.3: Use `accumulate` to sum the elements in a `vector<int>`.

Exercise 10.4: Assuming `v` is a `vector<double>`, what, if anything, is wrong with calling `accumulate(v.cbegin(), v.cend(), 0)`?

Exercise 10.5: In the call to `equal` on rosters, what would happen if both rosters held C-style strings, rather than library `strings`?

10.2.2. Algorithms That Write Container Elements

Some algorithms assign new values to the elements in a sequence. When we use an algorithm that assigns to elements, we must take care to ensure that the sequence into which the algorithm writes is at least as large as the number of elements we ask the algorithm to write. Remember, algorithms do not perform container operations, so they have no way themselves to change the size of a container.

Some algorithms write to elements in the input range itself. These algorithms are not inherently dangerous because they write only as many elements as are in the specified range.

As one example, the `fill` algorithm takes a pair of iterators that denote a range and a third argument that is a value. `fill` assigns the given value to each element in the input sequence:

Key Concept: Iterator Arguments

Some algorithms read elements from two sequences. The elements that constitute these sequences can be stored in different kinds of containers. For

example, the first sequence might be stored in a `vector` and the second might be in a `list`, a `deque`, a built-in array, or some other sequence. Moreover, the element types in the two sequences are not required to match exactly. What is required is that we be able to compare elements from the two sequences. For example, in the `equal` algorithm, the element types need not be identical, but we do have to be able to use == to compare elements from the two sequences.

Algorithms that operate on two sequences differ as to how we pass the second sequence. Some algorithms, such as `equal`, take three iterators: The first two denote the range of the first sequence, and the third iterator denotes the first element in the second sequence. Others take four iterators: The first two denote the range of elements in the first sequence, and the second two denote the range for the second sequence.

Algorithms that use a single iterator to denote the second sequence *assume* that the second sequence is at least as large as the first. It is up to us to ensure that the algorithm will not attempt to access a nonexistent element in the second sequence. For example, the `equal` algorithm potentially compares every element from its first sequence to an element in the second. If the second sequence is a subset of the first, then our program has a serious error—`equal` will attempt to access elements beyond the end of the second sequence.

Click here to view code image

```
fill(vec.begin(), vec.end(), 0);  // reset each element to 0
// set a subsequence of the container to 10
fill(vec.begin(), vec.begin() + vec.size()/2, 10);
```

Because `fill` writes to its given input sequence, so long as we pass a valid input sequence, the writes will be safe.

Algorithms Do Not Check Write Operations

Some algorithms take an iterator that denotes a separate destination. These algorithms assign new values to the elements of a sequence starting at the element denoted by the destination iterator. For example, the `fill_n` function takes a single iterator, a count, and a value. It assigns the given value to the specified number of elements starting at the element denoted to by the iterator. We might use `fill_n` to assign a new value to the elements in a `vector`:

Click here to view code image

```
vector<int> vec;  // empty vector
```

```
// use vec giving it various values
fill_n(vec.begin(), vec.size(), 0); // reset all the elements of vec to 0
```

The `fill_n` function assumes that it is safe to write the specified number of elements. That is, for a call of the form

```
fill_n(dest, n, val)
```

`fill_n` assumes that `dest` refers to an element and that there are at least `n` elements in the sequence starting from `dest`.

It is a fairly common beginner mistake to call `fill_n` (or similar algorithms that write to elements) on a container that has no elements:

Click here to view code image

```
vector<int> vec;  // empty vector
// disaster: attempts to write to ten (nonexistent) elements in vec
fill_n(vec.begin(), 10, 0);
```

This call to `fill_n` is a disaster. We specified that ten elements should be written, but there are no such elements—`vec` is empty. The result is undefined.

Warning

Algorithms that write to a destination iterator *assume* the destination is large enough to hold the number of elements being written.

Introducing back_inserter

One way to ensure that an algorithm has enough elements to hold the output is to use an **insert iterator**. An insert iterator is an iterator that *adds* elements to a container. Ordinarily, when we assign to a container element through an iterator, we assign to the element that iterator denotes. When we assign through an insert iterator, a new element equal to the right-hand value is added to the container.

We'll have more to say about insert iterators in § 10.4.1 (p. 401). However, in order to illustrate how to use algorithms that write to a container, we will use **back_inserter**, which is a function defined in the `iterator` header.

`back_inserter` takes a reference to a container and returns an insert iterator bound to that container. When we assign through that iterator, the assignment calls `push_back` to add an element with the given value to the container:

Click here to view code image

```
vector<int> vec; // empty vector
```

```
auto it = back_inserter(vec); // assigning through it adds elements to vec
*it = 42;                     // vec now has one element with value 42
```

We frequently use `back_inserter` to create an iterator to use as the destination of an algorithm. For example:

Click here to view code image

```
vector<int> vec; // empty vector
// ok: back_inserter creates an insert iterator that adds elements to vec
fill_n(back_inserter(vec), 10, 0);  // appends ten elements to vec
```

On each iteration, `fill_n` assigns to an element in the given sequence. Because we passed an iterator returned by `back_inserter`, each assignment will call `push_back` on `vec`. As a result, this call to `fill_n` adds ten elements to the end of `vec`, each of which has the value `0`.

Copy Algorithms

The `copy` algorithm is another example of an algorithm that writes to the elements of an output sequence denoted by a destination iterator. This algorithm takes three iterators. The first two denote an input range; the third denotes the beginning of the destination sequence. This algorithm copies elements from its input range into elements in the destination. It is essential that the destination passed to `copy` be at least as large as the input range.

As one example, we can use `copy` to copy one built-in array to another:

Click here to view code image

```
int a1[] = {0,1,2,3,4,5,6,7,8,9};
int a2[sizeof(a1)/sizeof(*a1)];  // a2 has the same size as a1
// ret points just past the last element copied into a2
auto ret = copy(begin(a1), end(a1), a2);  // copy a1 into a2
```

Here we define an array named `a2` and use `sizeof` to ensure that `a2` has as many elements as the array `a1` (§ 4.9, p. 157). We then call `copy` to copy `a1` into `a2`. After the call to `copy`, the elements in both arrays have the same values.

The value returned by `copy` is the (incremented) value of its destination iterator. That is, `ret` will point just past the last element copied into `a2`.

Several algorithms provide so-called “copying” versions. These algorithms compute new element values, but instead of putting them back into their input sequence, the algorithms create a new sequence to contain the results.

For example, the `replace` algorithm reads a sequence and replaces every instance of a given value with another value. This algorithm takes four parameters: two iterators denoting the input range, and two values. It replaces each element that is

equal to the first value with the second:

Click here to view code image

```
// replace any element with the value 0 with 42
replace(ilst.begin(), ilst.end(), 0, 42);
```

This call replaces all instances of `0` by `42`. If we want to leave the original sequence unchanged, we can call `replace_copy`. That algorithm takes a third iterator argument denoting a destination in which to write the adjusted sequence:

Click here to view code image

```
// use back_inserter to grow destination as needed
replace_copy(ilst.cbegin(), ilst.cend(),
             back_inserter(ivec), 0, 42);
```

After this call, `ilst` is unchanged, and `ivec` contains a copy of `ilst` with the exception that every element in `ilst` with the value 0 has the value 42 in `ivec`.

10.2.3. Algorithms That Reorder Container Elements

Some algorithms rearrange the order of elements within a container. An obvious example of such an algorithm is `sort`. A call to `sort` arranges the elements in the input range into sorted order using the element type's < operator.

As an example, suppose we want to analyze the words used in a set of children's stories. We'll assume that we have a `vector` that holds the text of several stories. We'd like to reduce this `vector` so that each word appears only once, regardless of how many times that word appears in any of the given stories.

For purposes of illustration, we'll use the following simple story as our input:

Click here to view code image

```
the quick red fox jumps over the slow red turtle
```

Given this input, our program should produce the following `vector`:

fox	jumps	over	quick	red	slow	the	turtle

Exercises Section 10.2.2

Exercise 10.6: Using `fill_n`, write a program to set a sequence of `int` values to 0.

Exercise 10.7: Determine if there are any errors in the following programs and, if so, correct the error(s):

(a)

Click here to view code image

```
vector<int> vec; list<int> lst; int i;
 while (cin >> i)
     lst.push_back(i);
 copy(lst.cbegin(), lst.cend(), vec.begin());
```

(b)

Click here to view code image

```
vector<int> vec;
 vec.reserve(10); // reserve is covered in § 9.4 (p. 356)
 fill_n(vec.begin(), 10, 0);
```

Exercise 10.8: We said that algorithms do not change the size of the containers over which they operate. Why doesn't the use of `back_inserter` invalidate this claim?

Eliminating Duplicates

To eliminate the duplicated words, we will first sort the `vector` so that duplicated words appear adjacent to each other. Once the `vector` is sorted, we can use another library algorithm, named `unique`, to reorder the `vector` so that the unique elements appear in the first part of the `vector`. Because algorithms cannot do container operations, we'll use the `erase` member of `vector` to actually remove the elements:

Click here to view code image

```
void elimDups(vector<string> &words)
{
     // sort words alphabetically so we can find the duplicates
     sort(words.begin(), words.end());
     // unique reorders the input range so that each word appears once in the
     // front portion of the range and returns an iterator one past the unique range
     auto end_unique = unique(words.begin(), words.end());
     // erase uses a vector operation to remove the nonunique elements
     words.erase(end_unique, words.end());
}
```

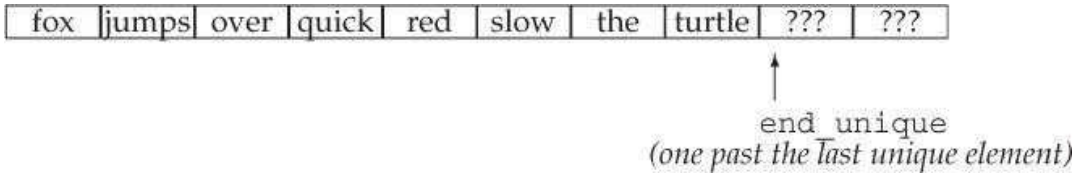
The `sort` algorithm takes two iterators denoting the range of elements to sort. In this call, we sort the entire `vector`. After the call to `sort`, `words` is ordered as

fox	jumps	over	quick	red	red	slow	the	the	turtle

Note that the words `red` and `the` appear twice.

Using unique

Once `words` is sorted, we want to keep only one copy of each word. The `unique` algorithm rearranges the input range to "eliminate" adjacent duplicated entries, and returns an iterator that denotes the end of the range of the unique values. After the call to `unique`, the `vector` holds



The size of `words` is unchanged; it still has ten elements. The order of those elements is changed—the adjacent duplicates have been "removed." We put remove in quotes because `unique` doesn't remove any elements. Instead, it overwrites adjacent duplicates so that the unique elements appear at the front of the sequence. The iterator returned by `unique` denotes one past the last unique element. The elements beyond that point still exist, but we don't know what values they have.

Note

The library algorithms operate on iterators, not containers. Therefore, an algorithm cannot (directly) add or remove elements.

Using Container Operations to Remove Elements

To actually remove the unused elements, we must use a container operation, which we do in the call to `erase` (§ 9.3.3, p. 349). We erase the range of elements from the one to which `end_unique` refers through the end of `words`. After this call, `words` contains the eight unique words from the input.

It is worth noting that this call to `erase` would be safe even if `words` has no duplicated words. In that case, `unique` would return `words.end()`. Both arguments to `erase` would have the same value: `words.end()`. The fact that the iterators are equal would mean that the range passed to `erase` would be empty. Erasing an empty range has no effect, so our program is correct even if the input has no duplicates.

Exercises Section 10.2.3

Exercise 10.9: Implement your own version of `elimDups`. Test your program by printing the `vector` after you read the input, after the call to `unique`, and after the call to `erase`.

Exercise 10.10: Why do you think the algorithms don't change the size of

containers?

10.3. Customizing Operations

Many of the algorithms compare elements in the input sequence. By default, such algorithms use either the element type's < or == operator. The library also defines versions of these algorithms that let us supply our own operation to use in place of the default operator.

For example, the `sort` algorithm uses the element type's < operator. However, we might want to sort a sequence into a different order from that defined by <, or our sequence might have elements of a type (such as `Sales_data`) that does not have a < operator. In both cases, we need to override the default behavior of `sort`.

10.3.1. Passing a Function to an Algorithm

As one example, assume that we want to print the `vector` after we call `elimDups` (§ 10.2.3, p. 384). However, we'll also assume that we want to see the words ordered by their size, and then alphabetically within each size. To reorder the `vector` by length, we'll use a second, overloaded version of `sort`. This version of `sort` takes a third argument that is a **predicate**.

Predicates

A predicate is an expression that can be called and that returns a value that can be used as a condition. The predicates used by library algorithms are either **unary predicates** (meaning they have a single parameter) or **binary predicates** (meaning they have two parameters). The algorithms that take predicates call the given predicate on the elements in the input range. As a result, it must be possible to convert the element type to the parameter type of the predicate.

The version of `sort` that takes a binary predicate uses the given predicate in place of < to compare elements. The predicates that we supply to `sort` must meet the requirements that we'll describe in § 11.2.2 (p. 425). For now, what we need to know is that the operation must define a consistent order for all possible elements in the input sequence. Our `isShorter` function from § 6.2.2 (p. 211) is an example of a function that meets these requirements, so we can pass `isShorter` to `sort`. Doing so will reorder the elements by size:

Click here to view code image

```
// comparison function to be used to sort by word length
bool isShorter(const string &s1, const string &s2)
```

```
{
    return s1.size() < s2.size();
}
// sort on word length, shortest to longest
sort(words.begin(), words.end(), isShorter);
```

If `words` contains the same data as in § 10.2.3 (p. 384), this call would reorder `words` so that all the words of length 3 appear before words of length 4, which in turn are followed by words of length 5, and so on.

Sorting Algorithms

When we sort `words` by size, we also want to maintain alphabetic order among the elements that have the same length. To keep the words of the same length in alphabetical order we can use the `stable_sort` algorithm. A stable sort maintains the original order among equal elements.

Ordinarily, we don't care about the relative order of equal elements in a sorted sequence. After all, they're equal. However, in this case, we have defined "equal" to mean "have the same length." Elements that have the same length still differ from one another when we view their contents. By calling `stable_sort`, we can maintain alphabetical order among those elements that have the same length:

Click here to view code image

```
elimDups(words); // put words in alphabetical order and remove duplicates
// resort by length, maintaining alphabetical order among words of the same length
stable_sort(words.begin(), words.end(), isShorter);
for (const auto &s : words)  // no need to copy the strings
    cout << s << " ";  // print each element separated by a space
cout << endl;
```

Assuming `words` was in alphabetical order before this call, after the call, `words` will be sorted by element size, and the words of each length remain in alphabetical order. If we run this code on our original `vector`, the output will be

fox red the over slow jumps quick turtle

Exercises Section 10.3.1

Exercise 10.11: Write a program that uses `stable_sort` and `isShorter` to sort a `vector` passed to your version of `elimDups`. Print the `vector` to verify that your program is correct.

Exercise 10.12: Write a function named `compareIsbn` that compares the `isbn()` members of two `Sales_data` objects. Use that function to `sort` a `vector` that holds `Sales_data` objects.

Exercise 10.13: The library defines an algorithm named `partition` that takes a predicate and partitions the container so that values for which the

predicate is `true` appear in the first part and those for which the predicate is `false` appear in the second part. The algorithm returns an iterator just past the last element for which the predicate returned `true`. Write a function that takes a `string` and returns a `bool` indicating whether the `string` has five characters or more. Use that function to partition `words`. Print the elements that have five or more characters.

10.3.2. Lambda Expressions

The predicates we pass to an algorithm must have exactly one or two parameters, depending on whether the algorithm takes a unary or binary predicate, respectively. However, sometimes we want to do processing that requires more arguments than the algorithm's predicate allows. For example, the solution you wrote for the last exercise in the previous section had to hard-wire the size 5 into the predicate used to partition the sequence. It would be move useful to be able to partition a sequence without having to write a separate predicate for every possible size.

As a related example, we'll revise our program from § 10.3.1 (p. 387) to report how many words are of a given size or greater. We'll also change the output so that it prints only the words of the given length or greater.

A sketch of this function, which we'll name `biggies`, is as follows:

Click here to view code image

```
void biggies(vector<string> &words,
             vector<string>::size_type sz)
{
     elimDups(words); // put words in alphabetical order and remove duplicates
     // resort by length, maintaining alphabetical order among words of the same length
     stable_sort(words.begin(), words.end(), isShorter);
     // get an iterator to the first element whose size() is >= sz
     // compute the number of elements with size >= sz
     // print words of the given size or longer, each one followed by a space
}
```

Our new problem is to find the first element in the `vector` that has the given size. Once we know that element, we can use its position to compute how many elements have that size or greater.

We can use the library `find_if` algorithm to find an element that has a particular size. Like `find` (§ 10.1, p. 376), the `find_if` algorithm takes a pair of iterators denoting a range. Unlike `find`, the third argument to `find_if` is a predicate. The `find_if` algorithm calls the given predicate on each element in the input range. It

returns the first element for which the predicate returns a nonzero value, or its end iterator if no such element is found.

It would be easy to write a function that takes a `string` and a size and returns a `bool` indicating whether the size of a given `string` is greater than the given size. However, `find_if` takes a unary predicate—any function we pass to `find_if` must have exactly one parameter that can be called with an element from the input sequence. There is no way to pass a second argument representing the size. To solve this part of our problem we'll need to use some additional language facilities.

Introducing Lambdas

We can pass any kind of **callable object** to an algorithm. An object or expression is callable if we can apply the call operator (§ 1.5.2, p. 23) to it. That is, if `e` is a callable expression, we can write `e(args)` where args is a comma-separated list of zero or more arguments.

The only callables we've used so far are functions and function pointers (§ 6.7, p. 247). There are two other kinds of callables: classes that overload the function-call operator, which we'll cover in § 14.8 (p. 571), and **lambda expressions**.

A lambda expression represents a callable unit of code. It can be thought of as an unnamed, inline function. Like any function, a lambda has a return type, a parameter list, and a function body. Unlike a function, lambdas may be defined inside a function. A lamba expression has the form

Click here to view code image

```
[capture list] (parameter list) -> return type { function body }
```

where *capture list* is an (often empty) list of local variables defined in the enclosing function; *return type*, *parameter list*, and *function body* are the same as in any ordinary function. However, unlike ordinary functions, a lambda must use a trailing return (§ 6.3.3, p. 229) to specify its return type.

We can omit either or both of the parameter list and return type but must always include the capture list and function body:

```
auto f = [] { return 42; };
```

Here, we've defined `f` as a callable object that takes no arguments and returns `42`.

We call a lambda the same way we call a function by using the call operator:

Click here to view code image

```
cout << f() << endl;  // prints 42
```

Omitting the parentheses and the parameter list in a lambda is equivalent to specifying an empty parameter list. Hence, when we call `f`, the argument list is

empty. If we omit the return type, the lambda has an inferred return type that depends on the code in the function body. If the function body is just a `return` statement, the return type is inferred from the type of the expression that is returned. Otherwise, the return type is `void`.

Note

Lambdas with function bodies that contain anything other than a single `return` statement that do not specify a return type return `void`.

Passing Arguments to a Lambda

As with an ordinary function call, the arguments in a call to a lambda are used to initialize the lambda's parameters. As usual, the argument and parameter types must match. Unlike ordinary functions, a lambda may not have default arguments (§ 6.5.1, p. 236). Therefore, a call to a lambda always has as many arguments as the lambda has parameters. Once the parameters are initialized, the function body executes.

As an example of a lambda that takes arguments, we can write a lambda that behaves like our `isShorter` function:

Click here to view code image

```
[](const string &a, const string &b)
  { return a.size() < b.size();}
```

The empty capture list indicates that this lambda will not use any local variables from the surrounding function. The lambda's parameters, like the parameters to `isShorter`, are references to `const string`. Again like `isShorter`, the lambda's function body compares its parameters' `size()`s and returns a `bool` that depends on the relative sizes of the given arguments.

We can rewrite our call to `stable_sort` to use this lambda as follows:

Click here to view code image

```
// sort words by size, but maintain alphabetical order for words of the same size
stable_sort(words.begin(), words.end(),
            [](const string &a, const string &b)
              { return a.size() < b.size();});
```

When `stable_sort` needs to compare two elements, it will call the given lambda expression.

Using the Capture List

We're now ready to solve our original problem, which is to write a callable expression

that we can pass to `find_if`. We want an expression that will compare the length of each `string` in the input sequence with the value of the `sz` parameter in the `biggies` function.

Although a lambda may appear inside a function, it can use variables local to that function *only* if it specifies which variables it intends to use. A lambda specifies the variables it will use by including those local variables in its capture list. The capture list directs the lambda to include information needed to access those variables within the lambda itself.

In this case, our lambda will capture `sz` and will have a single `string` parameter. The body of our lambda will compare the given `string`'s size with the captured value of `sz`:

Click here to view code image

```
[sz](const string &a)
    { return a.size() >= sz; };
```

Inside the `[]` that begins a lambda we can provide a comma-separated list of names defined in the surrounding function.

Because this lambda captures `sz`, the body of the lambda may use `sz`. The lambda does not capture `words`, and so has no access to that variable. Had we given our lambda an empty capture list, our code would not compile:

Click here to view code image

```
// error: sz not captured
[](const string &a)
    { return a.size() >= sz; };
```

Note

A lambda may use a variable local to its surrounding function *only* if the lambda captures that variable in its capture list.

Calling find_if

Using this lambda, we can find the first element whose size is at least as big as `sz`:

Click here to view code image

```
// get an iterator to the first element whose size() is >= sz
auto wc = find_if(words.begin(), words.end(),
            [sz](const string &a)
                { return a.size() >= sz; });
```

The call to `find_if` returns an iterator to the first element that is at least as long as

the given sz, or a copy of words.end() if no such element exists.

We can use the iterator returned from find_if to compute how many elements appear between that iterator and the end of words (§ 3.4.2, p. 111):

Click here to view code image

```
// compute the number of elements with size >= sz
auto count = words.end() - wc;
cout << count << " " << make_plural(count, "word", "s")
     << " of length " << sz << " or longer" << endl;
```

Our output statement calls make_plural (§ 6.3.2, p. 224) to print word or words, depending on whether that size is equal to 1.

The for_each Algorithm

The last part of our problem is to print the elements in words that have length sz or greater. To do so, we'll use the for_each algorithm. This algorithm takes a callable object and calls that object on each element in the input range:

Click here to view code image

```
// print words of the given size or longer, each one followed by a space
for_each(wc, words.end(),
         [](const string &s){cout << s << " ";});
cout << endl;
```

The capture list in this lambda is empty, yet the body uses two names: its own parameter, named s, and cout.

The capture list is empty, because we use the capture list only for (nonstatic) variables defined in the surrounding function. A lambda can use names that are defined outside the function in which the lambda appears. In this case, cout is not a name defined locally in biggies; that name is defined in the iostream header. So long as the iostream header is included in the scope in which biggies appears, our lambda can use cout.

Note

The capture list is used for local nonstatic variables only; lambdas can use local statics and variables declared outside the function directly.

Putting It All Together

Now that we've looked at the program in detail, here is the program as a whole:

Click here to view code image

```
void biggies(vector<string> &words,
             vector<string>::size_type sz)
{
     elimDups(words); // put words in alphabetical order and remove duplicates
     // sort words by size, but maintain alphabetical order for words of the same size
     stable_sort(words.begin(), words.end(),
                 [](const string &a, const string &b)
                   { return a.size() < b.size();});
     // get an iterator to the first element whose size() is >= sz
     auto wc = find_if(words.begin(), words.end(),
                 [sz](const string &a)
                     { return a.size() >= sz; });
     // compute the number of elements with size >= sz
     auto count = words.end() - wc;
     cout << count << " " << make_plural(count, "word", "s")
          << " of length " << sz << " or longer" << endl;
     // print words of the given size or longer, each one followed by a space
     for_each(wc, words.end(),
              [](const string &s){cout << s << " ";});
     cout << endl;
}
```

Exercises Section 10.3.2

Exercise 10.14: Write a lambda that takes two `int`s and returns their sum.

Exercise 10.15: Write a lambda that captures an `int` from its enclosing function and takes an `int` parameter. The lambda should return the sum of the captured `int` and the `int` parameter.

Exercise 10.16: Write your own version of the `biggies` function using lambdas.

Exercise 10.17: Rewrite exercise 10.12 from § 10.3.1 (p. 387) to use a lambda in the call to `sort` instead of the `compareIsbn` function.

Exercise 10.18: Rewrite `biggies` to use `partition` instead of `find_if`. We described the `partition` algorithm in exercise 10.13 in § 10.3.1 (p. 387).

Exercise 10.19: Rewrite the previous exercise to use `stable_partition`, which like `stable_sort` maintains the original element order in the paritioned sequence.

10.3.3. Lambda Captures and Returns

When we define a lambda, the compiler generates a new (unnamed) class type that

corresponds to that lambda. We'll see how these classes are generated in § 14.8.1 (p. 572). For now, what's useful to understand is that when we pass a lambda to a function, we are defining both a new type and an object of that type: The argument is an unnamed object of this compiler-generated class type. Similarly, when we use `auto` to define a variable initialized by a lambda, we are defining an object of the type generated from that lambda.

By default, the class generated from a lambda contains a data member corresponding to the variables captured by the lambda. Like the data members of any class, the data members of a lambda are initialized when a lambda object is created.

Capture by Value

Similar to parameter passing, we can capture variables by value or by reference. Table 10.1 (p. 395) covers the various ways we can form a capture list. So far, our lambdas have captured variables by value. As with a parameter passed by value, it must be possible to copy such variables. Unlike parameters, the value of a captured variable is copied when the lambda is created, not when it is called:

Click here to view code image

```
void fcn1()
{
    size_t v1 = 42;   // local variable
    // copies v1 into the callable object named f
    auto f = [v1] { return v1; };
    v1 = 0;
    auto j = f(); // j is 42; f stored a copy of v1 when we created it
}
```

Table 10.1. Lambda Capture List

`[]`	Empty capture list. The lambda may not use variables from the enclosing function. A lamba may use local variables only if it captures them.
`[`*names*`]`	*names* is a comma-separated list of names local to the enclosing function. By default, variables in the capture list are copied. A name preceded by `&` is captured by reference.
`[&]`	Implicit by reference capture list. Entities from the enclosing function used in the lambda body are used by reference.
`[=]`	Implicit by value capture list. Entities from the enclosing function used in the lambda body are copied into the lambda body.
`[&,` *identifier_list*`]`	*identifier_list* is a comma-separated list of zero or more variables from the enclosing function. These variables are captured by value; any implicitly captured variables are captured by reference. The names in *identifier_list* must not be preceded by an `&`.
`[=,` *reference_list*`]`	Variables included in the *reference_list* are captured by reference; any implicitly captured variables are captured by value. The names in *reference_list* may not include `this` and must be preceded by an `&`.

Because the value is copied when the lambda is created, subsequent changes to a captured variable have no effect on the corresponding value inside the lambda.

Capture by Reference

We can also define lambdas that capture variables by reference. For example:

Click here to view code image

```
void fcn2()
{
    size_t v1 = 42;   // local variable
    // the object f2 contains a reference to v1
    auto f2 = [&v1] { return v1; };
    v1 = 0;
    auto j = f2(); // j is 0; f2 refers to v1; it doesn't store it
}
```

The `&` before `v1` indicates that `v1` should be captured as a reference. A variable captured by reference acts like any other reference. When we use the variable inside the lambda body, we are using the object to which that reference is bound. In this case, when the lambda returns `v1`, it returns the value of the object to which `v1` refers.

Reference captures have the same problems and restrictions as reference returns (§ 6.3.2, p. 225). If we capture a variable by reference, we must be *certain* that the referenced object exists at the time the lambda is executed. The variables captured by a lambda are local variables. These variables cease to exist once the function completes. If it is possible for a lambda to be executed after the function finishes, the local variables to which the capture refers no longer exist.

Reference captures are sometimes necessary. For example, we might want our `biggies` function to take a reference to an `ostream` on which to write and a character to use as the separator:

Click here to view code image

```
void biggies(vector<string> &words,
             vector<string>::size_type sz,
             ostream &os = cout, char c = ' ')
{
    // code to reorder words as before
    // statement to print count revised to print to os
    for_each(words.begin(), words.end(),
             [&os, c](const string &s) { os << s << c; });
}
```

We cannot copy `ostream` objects (§ 8.1.1, p. 311); the only way to capture `os` is by reference (or through a pointer to `os`).

When we pass a lambda to a function, as in this call to `for_each`, the lambda executes immediately. Capturing `os` by reference is fine, because the variables in `biggies` exist while `for_each` is running.

We can also return a lambda from a function. The function might directly return a callable object or the function might return an object of a class that has a callable object as a data member. If the function returns a lambda, then—for the same reasons that a function must not return a reference to a local variable—that lambda must not contain reference captures.

Warning

When we capture a variable by reference, we must ensure that the variable exists at the time that the lambda executes.

Advice: Keep Your Lambda Captures Simple

A lambda capture stores information between the time the lambda is created (i.e., when the code that defines the lambda is executed) and the time (or times) the lambda itself is executed. It is the programmer's responsibility to ensure that whatever information is captured has the intended meaning each time the lambda is executed.

Capturing an ordinary variable—an `int`, a `string`, or other nonpointer type—by value is usually straightforward. In this case, we only need to care whether the variable has the value we need when we capture it.

If we capture a pointer or iterator, or capture a variable by reference, we must ensure that the object bound to that iterator, pointer, or reference still

exists, whenever the lambda *executes*. Moreover, we need to ensure that the object has the intended value. Code that executes between when the lambda is created and when it executes might change the value of the object to which the lambda capture points (or refers). The value of the object at the time the pointer (or reference) was captured might have been what we wanted. The value of that object when the lambda executes might be quite different.

As a rule, we can avoid potential problems with captures by minimizing the data we capture. Moreover, if possible, avoid capturing pointers or references.

Implicit Captures

Rather than explicitly listing the variables we want to use from the enclosing function, we can let the compiler infer which variables we use from the code in the lambda's body. To direct the compiler to infer the capture list, we use an & or = in the capture list. The & tells the compiler to capture by reference, and the = says the values are captured by value. For example, we can rewrite the lambda that we passed to `find_if` as

Click here to view code image

```
// sz implicitly captured by value
wc = find_if(words.begin(), words.end(),
             [=](const string &s)
                { return s.size() >= sz; });
```

If we want to capture some variables by value and others by reference, we can mix implicit and explicit captures:

Click here to view code image

```
void biggies(vector<string> &words,
             vector<string>::size_type sz,
             ostream &os = cout, char c = ' ')
{
    // other processing as before
    // os implicitly captured by reference; c explicitly captured by value
    for_each(words.begin(), words.end(),
             [&, c](const string &s) { os << s << c; });
    // os explicitly captured by reference; c implicitly captured by value
    for_each(words.begin(), words.end(),
             [=, &os](const string &s) { os << s << c; });
}
```

When we mix implicit and explicit captures, the first item in the capture list must be an & or =. That symbol sets the default capture mode as by reference or by value,

respectively.

When we mix implicit and explicit captures, the explicitly captured variables must use the alternate form. That is, if the implicit capture is by reference (using `&`), then the explicitly named variables must be captured by value; hence their names may not be preceded by an `&`. Alternatively, if the implicit capture is by value (using `=`), then the explicitly named variables must be preceded by an `&` to indicate that they are to be captured by reference.

Mutable Lambdas

By default, a lambda may not change the value of a variable that it copies by value. If we want to be able to change the value of a captured variable, we must follow the parameter list with the keyword `mutable`. Lambdas that are mutable may not omit the parameter list:

Click here to view code image

```
void fcn3()
{
    size_t v1 = 42; // local variable
    // f can change the value of the variables it captures
    auto f = [v1] () mutable { return ++v1; };
    v1 = 0;
    auto j = f(); // j is 43
}
```

Whether a variable captured by reference can be changed (as usual) depends only on whether that reference refers to a `const` or nonconst type:

Click here to view code image

```
void fcn4()
{
    size_t v1 = 42;   // local variable
    // v1 is a reference to a non const variable
    // we can change that variable through the reference inside f2
    auto f2 = [&v1] { return ++v1; };
    v1 = 0;
    auto j = f2(); // j is 1
}
```

Specifying the Lambda Return Type

The lambdas we've written so far contain only a single `return` statement. As a result, we haven't had to specify the return type. By default, if a lambda body contains any statements other than a `return`, that lambda is assumed to return `void`. Like other functions that return `void`, lambdas inferred to return `void` may not return a value.

As a simple example, we might use the library `transform` algorithm and a lambda to replace each negative value in a sequence with its absolute value:

Click here to view code image

```
transform(vi.begin(), vi.end(), vi.begin(),
          [](int i) { return i < 0 ? -i : i; });
```

The `transform` function takes three iterators and a callable. The first two iterators denote an input sequence and the third is a destination. The algorithm calls the given callable on each element in the input sequence and writes the result to the destination. As in this call, the destination iterator can be the same as the iterator denoting the start of the input. When the input iterator and the destination iterator are the same, `transform` replaces each element in the input range with the result of calling the callable on that element.

In this call, we passed a lambda that returns the absolute value of its parameter. The lambda body is a single `return` statement that returns the result of a conditional expression. We need not specify the return type, because that type can be inferred from the type of the conditional operator.

However, if we write the seemingly equivalent program using an `if` statement, our code won't compile:

Click here to view code image

```
// error: cannot deduce the return type for the lambda
transform(vi.begin(), vi.end(), vi.begin(),
            [](int i) { if (i < 0) return -i; else return i;
});
```

This version of our lambda infers the return type as `void` but we returned a value.

When we need to define a return type for a lambda, we must use a trailing return type (§ 6.3.3, p. 229):

Click here to view code image

```
transform(vi.begin(), vi.end(), vi.begin(),
          [](int i) -> int
          { if (i < 0) return -i; else return i; });
```

In this case, the fourth argument to `transform` is a lambda with an empty capture list, which takes a single parameter of type `int` and returns a value of type `int`. Its function body is an `if` statement that returns the absolute value of its parameter.

Exercises Section 10.3.3

Exercise 10.20: The library defines an algorithm named `count_if`. Like `find_if`, this function takes a pair of iterators denoting an input range and

a predicate that it applies to each element in the given range. `count_if` returns a count of how often the predicate is true. Use `count_if` to rewrite the portion of our program that counted how many words are greater than length 6.

Exercise 10.21: Write a lambda that captures a local `int` variable and decrements that variable until it reaches 0. Once the variable is 0 additional calls should no longer decrement the variable. The lambda should return a `bool` that indicates whether the captured variable is 0.

10.3.4. Binding Arguments

Lambda expressions are most useful for simple operations that we do not need to use in more than one or two places. If we need to do the same operation in many places, we should usually define a function rather than writing the same lambda expression multiple times. Similarly, if an operation requires many statements, it is ordinarily better to use a function.

It is usually straightforward to use a function in place of a lambda that has an empty capture list. As we've seen, we can use either a lambda or our `isShorter` function to order the `vector` on word length. Similarly, it would be easy to replace the lambda that printed the contents of our `vector` by writing a function that takes a `string` and prints the given `string` to the standard output.

However, it is not so easy to write a function to replace a lambda that captures local variables. For example, the lambda that we used in the call to `find_if` compared a `string` with a given size. We can easily write a function to do the same work:

Click here to view code image

```
bool check_size(const string &s, string::size_type sz)
{
    return s.size() >= sz;
}
```

However, we can't use this function as an argument to `find_if`. As we've seen, `find_if` takes a unary predicate, so the callable passed to `find_if` must take a single argument. The lambda that `biggies` passed to `find_if` used its capture list to store `sz`. In order to use `check_size` in place of that lambda, we have to figure out how to pass an argument to the `sz` parameter.

The Library bind Function

We can solve the problem of passing a size argument to `check_size` by using a new

library function named **bind**, which is defined in the `functional` header. The `bind` function can be thought of as a general-purpose function adaptor (§ 9.6, p. 368). It takes a callable object and generates a new callable that "adapts" the parameter list of the original object.

The general form of a call to `bind` is:

Click here to view code image

```
auto newCallable = bind(callable, arg_list);
```

where *newCallable* is itself a callable object and *arg_list* is a comma-separated list of arguments that correspond to the parameters of the given *callable*. That is, when we call *newCallable*, *newCallable* calls *callable*, passing the arguments in *arg_list*.

The arguments in *arg_list* may include names of the form *_n*, where *n* is an integer. These arguments are "placeholders" representing the parameters of *newCallable*. They stand "in place of" the arguments that will be passed to *newCallable*. The number *n* is the position of the parameter in the generated callable: `_1` is the first parameter in *newCallable*, `_2` is the second, and so forth.

Binding the sz Parameter of check_size

As a simple example, we'll use `bind` to generate an object that calls `check_size` with a fixed value for its size parameter as follows:

Click here to view code image

```
// check6 is a callable object that takes one argument of type string
// and calls check_size on its given string and the value 6
auto check6 = bind(check_size, _1, 6);
```

This call to `bind` has only one placeholder, which means that `check6` takes a single argument. The placeholder appears first in *arg_list*, which means that the parameter in `check6` corresponds to the first parameter of `check_size`. That parameter is a `const string&`, which means that the parameter in `check6` is also a `const string&`. Thus, a call to `check6` must pass an argument of type `string`, which `check6` will pass as the first argument to `check_size`.

The second argument in *arg_list* (i.e., the third argument to `bind`) is the value `6`. That value is bound to the second parameter of `check_size`. Whenever we call `check6`, it will pass `6` as the second argument to `check_size`:

Click here to view code image

```
string s = "hello";
bool b1 = check6(s);  // check6(s) calls check_size(s, 6)
```

Using `bind`, we can replace our original lambda-based call to `find_if`

Click here to view code image

```
auto wc = find_if(words.begin(), words.end(),
            [sz](const string &a)
```

with a version that uses `check_size`:

Click here to view code image

```
auto wc = find_if(words.begin(), words.end(),
              bind(check_size, _1, sz));
```

This call to `bind` generates a callable object that binds the second argument of `check_size` to the value of `sz`. When `find_if` calls this object on the `strings` in `words` those calls will in turn call `check_size` passing the given `string` and `sz`. So, `find_if` (effectively) will call `check_size` on each `string` in the input range and compare the size of that `string` to `sz`.

Using placeholders Names

The *_n* names are defined in a namespace named `placeholders`. That namespace is itself defined inside the `std` namespace (§ 3.1, p. 82). To use these names, we must supply the names of both namespaces. As with our other examples, our calls to `bind` assume the existence of appropriate `using` declarations. For example, the `using` declaration for `_1` is

Click here to view code image

```
using std::placeholders::_1;
```

This declaration says we're using the name `_1`, which is defined in the namespace `placeholders`, which is itself defined in the namespace `std`.

We must provide a separate `using` declaration for each placeholder name that we use. Writing such declarations can be tedious and error-prone. Rather than separately declaring each placeholder, we can use a different form of `using` that we will cover in more detail in § 18.2.2 (p. 793). This form:

Click here to view code image

```
using namespace namespace_name;
```

says that we want to make all the names from *namespace_name* accessible to our program. For example:

Click here to view code image

```
using namespace std::placeholders;
```

makes all the names defined by `placeholders` usable. Like the `bind` function, the `placeholders` namespace is defined in the `functional` header.

Arguments to bind

As we've seen, we can use `bind` to fix the value of a parameter. More generally, we can use `bind` to bind or rearrange the parameters in the given callable. For example, assuming `f` is a callable object that has five parameters, the following call to `bind`:

Click here to view code image

```
// g is a callable object that takes two arguments
auto g = bind(f, a, b, _2, c, _1);
```

generates a new callable that takes two arguments, represented by the placeholders `_2` and `_1`. The new callable will pass its own arguments as the third and fifth arguments to `f`. The first, second, and fourth arguments to `f` are bound to the given values, `a`, `b`, and `c`, respectively.

The arguments to `g` are bound positionally to the placeholders. That is, the first argument to `g` is bound to `_1`, and the second argument is bound to `_2`. Thus, when we call `g`, the first argument to `g` will be passed as the last argument to `f`; the second argument to `g` will be passed as `g`'s third argument. In effect, this call to `bind` maps

```
g(_1, _2)
```

to

```
f(a, b, _2, c, _1)
```

That is, calling `g` calls `f` using `g`'s arguments for the placeholders along with the bound arguments, `a`, `b`, and `c`. For example, calling `g(X, Y)` calls

```
f(a, b, Y, c, X)
```

Using to bind to Reorder Parameters

As a more concrete example of using `bind` to reorder arguments, we can use `bind` to invert the meaning of `isShorter` by writing

Click here to view code image

```
// sort on word length, shortest to longest
sort(words.begin(), words.end(), isShorter);
// sort on word length, longest to shortest
sort(words.begin(), words.end(), bind(isShorter, _2, _1));
```

In the first call, when `sort` needs to compare two elements, `A` and `B`, it will call `isShorter(A, B)`. In the second call to `sort`, the arguments to `isShorter` are swapped. In this case, when `sort` compares elements, it will be as if `sort` called `isShorter(B, A)`.

Binding Reference Parameters

By default, the arguments to `bind` that are not placeholders are copied into the

callable object that `bind` returns. However, as with lambdas, sometimes we have arguments that we want to bind but that we want to pass by reference or we might want to bind an argument that has a type that we cannot copy.

For example, to replace the lambda that captured an `ostream` by reference:

Click here to view code image

```
// os is a local variable referring to an output stream
// c is a local variable of type char
for_each(words.begin(), words.end(),
         [&os, c](const string &s) { os << s << c; });
```

We can easily write a function to do the same job:

Click here to view code image

```
ostream &print(ostream &os, const string &s, char c)
{
    return os << s << c;
}
```

However, we can't use `bind` directly to replace the capture of `os`:

Click here to view code image

```
// error: cannot copy os
for_each(words.begin(), words.end(), bind(print, os, _1, '
'));
```

because `bind` copies its arguments and we cannot copy an `ostream`. If we want to pass an object to `bind` without copying it, we must use the library **ref** function:

Click here to view code image

```
for_each(words.begin(), words.end(),
         bind(print, ref(os), _1, ' '));
```

The `ref` function returns an object that contains the given reference and that is itself copyable. There is also a **cref** function that generates a class that holds a reference to `const`. Like `bind`, the `ref` and `cref` functions are defined in the `functional` header.

Backward Compatibility: Binding Arguments

Older versions of C++ provided a much more limited, yet more complicated, set of facilities to bind arguments to functions. The library defined two functions named `bind1st` and `bind2nd`. Like `bind`, these functions take a function and generate a new callable object that calls the given function with one of its parameters bound to a given value. However, these functions can bind only the first or second parameter, respectively. Because they are of much more limited utility, they have been *deprecated* in the new standard. A deprecated feature is one that may not be supported in future releases.

Modern C++ programs should use `bind`.

Exercises Section 10.3.4

Exercise 10.22: Rewrite the program to count words of size 6 or less using functions in place of the lambdas.

Exercise 10.23: How many arguments does `bind` take?

Exercise 10.24: Use `bind` and `check_size` to find the first element in a `vector` of `int`s that has a value greater than the length of a specified `string` value.

Exercise 10.25: In the exercises for § 10.3.2 (p. 392) you wrote a version of `biggies` that uses `partition`. Rewrite that function to use `check_size` and `bind`.

10.4. Revisiting Iterators

In addition to the iterators that are defined for each of the containers, the library defines several additional kinds of iterators in the `iterator` header. These iterators include

- **Insert iterators**: These iterators are bound to a container and can be used to insert elements into the container.
- **Stream iterators**: These iterators are bound to input or output streams and can be used to iterate through the associated IO stream.
- **Reverse iterators**: These iterators move backward, rather than forward. The library containers, other than `forward_list`, have reverse iterators.
- **Move iterators**: These special-purpose iterators move rather than copy their elements. We'll cover move iterators in § 13.6.2 (p. 543).

10.4.1. Insert Iterators

An inserter is an iterator adaptor (§ 9.6, p. 368) that takes a container and yields an iterator that adds elements to the specified container. When we assign a value through an insert iterator, the iterator calls a container operation to add an element at a specified position in the given container. The operations these iterators support are listed in Table 10.2 (overleaf).

Table 10.2. Insert Iterator Operations

`it = t`	Inserts the value `t` at the current position denoted by `it`. Depending on the kind of insert iterator, and assuming `c` is the container to which `it` is bound, calls `c.push_back(t)`, `c.push_front(t)`, or `c.insert(t, p)`, where `p` is the iterator position given to `inserter`.
`*it, ++it, it++`	These operations exist but do nothing to `it`. Each operator returns `it`.

There are three kinds of inserters. Each differs from the others as to where elements are inserted:

- `back_inserter` (§ 10.2.2, p. 382) creates an iterator that uses `push_back`.
- **front_inserter** creates an iterator that uses `push_front`.
- **inserter** creates an iterator that uses `insert`. This function takes a second argument, which must be an iterator into the given container. Elements are inserted ahead of the element denoted by the given iterator.

Note

We can use `front_inserter` *only* if the container has `push_front`. Similarly, we can use `back_inserter` *only* if it has `push_back`.

It is important to understand that when we call `inserter(c, iter)`, we get an iterator that, when used successively, inserts elements ahead of the element originally denoted by `iter`. That is, if `it` is an iterator generated by `inserter`, then an assignment such as

```
* it = va1;
```

behaves as

Click here to view code image

```
it = c.insert(it, val); // it points to the newly added element
++it; // increment it so that it denotes the same element as before
```

The iterator generated by `front_inserter` behaves quite differently from the one created by `inserter`. When we use `front_inserter`, elements are always inserted ahead of the then first element in the container. Even if the position we pass to `inserter` initially denotes the first element, as soon as we insert an element in front of that element, that element is no longer the one at the beginning of the container:

Click here to view code image

```
list<int> 1st = {1,2,3,4};
list<int> lst2, lst3;      // empty lists
// after copy completes, 1st2 contains 4 3 2 1
```

```
copy(lst.cbegin(), lst.cend(), front_inserter(lst2));
// after copy completes, lst3 contains 1 2 3 4
copy(lst.cbegin(), lst.cend(), inserter(lst3, lst3.begin()));
```

When we call `front_inserter(c)`, we get an insert iterator that successively calls `push_front`. As each element is inserted, it becomes the new first element in `c`. Therefore, `front_inserter` yields an iterator that reverses the order of the sequence that it inserts; `inserter` and `back_inserter` don't.

Exercises Section 10.4.1

Exercise 10.26: Explain the differences among the three kinds of insert iterators.

Exercise 10.27: In addition to `unique` (§ 10.2.3, p. 384), the library defines function named `unique_copy` that takes a third iterator denoting a destination into which to copy the unique elements. Write a program that uses `unique_copy` to copy the unique elements from a `vector` into an initially empty `list`.

Exercise 10.28: Copy a `vector` that holds the values from `1` to `9` inclusive, into three other containers. Use an `inserter`, a `back_inserter`, and a `front_inserter`, respectivly to add elements to these containers. Predict how the output sequence varies by the kind of inserter and verify your predictions by running your programs.

10.4.2. iostream Iterators

Even though the `iostream` types are not containers, there are iterators that can be used with objects of the IO types (§ 8.1, p. 310). An **istream_iterator** (Table 10.3 (overleaf)) reads an input stream, and an **ostream_iterator** (Table 10.4 (p. 405)) writes an output stream. These iterators treat their corresponding stream as a sequence of elements of a specified type. Using a stream iterator, we can use the generic algorithms to read data from or write data to stream objects.

Table 10.3. istream_iterator Operations

`istream_iterator<T> in(is);`	`in` reads values of type `T` from input stream `is`.
`istream_iterator<T> end;`	Off-the-end iterator for an `istream_iterator` that reads values of type `T`.
`in1 == in2` `in1 != in2`	`in1` and `in2` must read the same type. They are equal if they are both the end value or are bound to the same input stream.
`*in`	Returns the value read from the stream.
`in->mem`	Synonym for `(*in).mem`.
`++in, in++`	Reads the next value from the input stream using the `>>` operator for the element type. As usual, the prefix version returns a reference to the incremented iterator. The postfix version returns the old value.

Table 10.4. ostream Iterator Operations

`ostream_iterator<T> out(os);`	`out` writes values of type `T` to output stream `os`.
`ostream_iterator<T> out(os, d);`	`out` writes values of type `T` followed by `d` to output stream `os`. `d` points to a null-terminated character array.
`out = val`	Writes `val` to the `ostream` to which `out` is bound using the `<<` operator. `val` must have a type that is compatible with the type that `out` can write.
`*out, ++out, out++`	These operations exist but do nothing to `out`. Each operator returns `out`.

Operations on istream_iterators

When we create a stream iterator, we must specify the type of objects that the iterator will read or write. An `istream_iterator` uses >> to read a stream. Therefore, the type that an `istream_iterator` reads must have an input operator defined. When we create an `istream_iterator`, we can bind it to a stream. Alternatively, we can default initialize the iterator, which creates an iterator that we can use as the off-the-end value.

Click here to view code image

```
istream_iterator<int> int_it(cin);    //    reads ints from cin
istream_iterator<int> int_eof;        //    end iterator value
ifstream in("afile");
istream_iterator<string> str_it(in); //    reads strings from "afile"
```

As an example, we can use an `istream_iterator` to read the standard input into a `vector`:

Click here to view code image

```
istream_iterator<int> in_iter(cin);  // read ints from cin
istream_iterator<int> eof;           // istream "end" iterator
```

```
while (in_iter != eof)  // while there's valid input to read
    // postfix increment reads the stream and returns the old value of the iterator
    // we dereference that iterator to get the previous value read from the stream
    vec.push_back(*in_iter++);
```

This loop reads `ints` from `cin`, storing what was read in `vec`. On each iteration, the loop checks whether `in_iter` is the same as `eof`. That iterator was defined as the empty `istream_iterator`, which is used as the end iterator. An iterator bound to a stream is equal to the end iterator once its associated stream hits end-of-file or encounters an IO error.

The hardest part of this program is the argument to `push_back`, which uses the dereference and postfix increment operators. This expression works just like others we've written that combined dereference with postfix increment (§ 4.5, p. 148). The postfix increment advances the stream by reading the next value but returns the *old* value of the iterator. That old value contains the previous value read from the stream. We dereference that iterator to obtain that value.

What is more useful is that we can rewrite this program as

Click here to view code image

```
istream_iterator<int> in_iter(cin), eof;  // read ints from cin
vector<int> vec(in_iter, eof);  // construct vec from an iterator range
```

Here we construct `vec` from a pair of iterators that denote a range of elements. Those iterators are `istream_iterators`, which means that the range is obtained by reading the associated stream. This constructor reads `cin` until it hits end-of-file or encounters an input that is not an `int`. The elements that are read are used to construct `vec`.

Using Stream Iterators with the Algorithms

Because algorithms operate in terms of iterator operations, and the stream iterators support at least some iterator operations, we can use stream iterators with at least some of the algorithms. We'll see in § 10.5.1 (p. 410) how to tell which algorithms can be used with the stream iterators. As one example, we can call `accumulate` with a pair of `istream_iterators`:

Click here to view code image

```
istream_iterator<int> in(cin), eof;
cout << accumulate(in, eof, 0) << endl;
```

This call will generate the sum of values read from the standard input. If the input to this program is

23 109 45 89 6 34 12 90 34 23 56 23 8 89 23

then the output will be `664`.

istream_iterators Are Permitted to Use Lazy Evaluation

When we bind an `istream_iterator` to a stream, we are not guaranteed that it will read the stream immediately. The implementation is permitted to delay reading the stream until we use the iterator. We are guaranteed that before we dereference the iterator for the first time, the stream will have been read. For most programs, whether the read is immediate or delayed makes no difference. However, if we create an `istream_iterator` that we destroy without using or if we are synchronizing reads to the same stream from two different objects, then we might care a great deal when the read happens.

Operations on ostream_iterators

An `ostream_iterator` can be defined for any type that has an output operator (the << operator). When we create an `ostream_iterator`, we may (optionally) provide a second argument that specifies a character string to print following each element. That string must be a C-style character string (i.e., a string literal or a pointer to a null-terminated array). We must bind an `ostream_iterator` to a specific stream. There is no empty or off-the-end `ostream_iterator`.

We can use an `ostream_iterator` to write a sequence of values:

Click here to view code image

```
ostream_iterator<int> out_iter(cout, " ");
for (auto e : vec)
    *out_iter++ = e;   // the assignment writes this element to cout
cout << endl;
```

This program writes each element from `vec` onto `cout` following each element with a space. Each time we assign a value to `out_iter`, the write is committed.

It is worth noting that we can omit the dereference and the increment when we assign to `out_iter`. That is, we can write this loop equivalently as

Click here to view code image

```
for (auto e : vec)
    out_iter = e;   // the assignment writes this element to cout
cout << endl;
```

The * and ++ operators do nothing on an `ostream_iterator`, so omitting them has no effect on our program. However, we prefer to write the loop as first presented. That loop uses the iterator consistently with how we use other iterator types. We can easily change this loop to execute on another iterator type. Moreover, the behavior of this loop will be clearer to readers of our code.

Rather than writing the loop ourselves, we can more easily print the elements in

vec by calling copy:

Click here to view code image

```
copy(vec.begin(), vec.end(), out_iter);
cout << endl;
```

Using Stream Iterators with Class Types

We can create an istream_iterator for any type that has an input operator (>>). Similarly, we can define an ostream_iterator so long as the type has an output operator (<<). Because Sales_item has both input and output operators, we can use IO iterators to rewrite the bookstore program from § 1.6 (p. 24):

Click here to view code image

```
istream_iterator<Sales_item> item_iter(cin), eof;
ostream_iterator<Sales_item> out_iter(cout, "\n");
// store the first transaction in sum and read the next record
Sales_item sum = *item_iter++;
while (item_iter != eof) {
    // if the current transaction (which is stored in item_iter) has the same ISBN
    if (item_iter->isbn() == sum.isbn())
        sum += *item_iter++; // add it to sum and read the next transaction
    else {
        out_iter = sum;       // write the current sum
        sum = *item_iter++;   // read the next transaction
    }
}
out_iter = sum;   // remember to print the last set of records
```

This program uses item_iter to read Sales_item transactions from cin. It uses out_iter to write the resulting sums to cout, following each output with a newline. Having defined our iterators, we use item_iter to initialize sum with the value of the first transaction:

Click here to view code image

```
// store the first transaction in sum and read the next record
Sales_item sum = *item_iter++;
```

Here, we dereference the result of the postfix increment on item_iter. This expression reads the next transaction, and initializes sum from the value previously stored in item_iter.

The while loop executes until we hit end-of-file on cin. Inside the while, we check whether sum and the record we just read refer to the same book. If so, we add the most recently read Sales_item into sum. If the ISBNs differ, we assign sum to out_iter, which prints the current value of sum followed by a newline. Having

printed the sum for the previous book, we assign `sum` a copy of the most recently read transaction and increment the iterator, which reads the next transaction. The loop continues until an error or end-of-file is encountered. Before exiting, we remember to print the values associated with the last book in the input.

Exercises Section 10.4.2

Exercise 10.29: Write a program using stream iterators to read a text file into a `vector` of `strings`.

Exercise 10.30: Use stream iterators, `sort`, and `copy` to read a sequence of integers from the standard input, sort them, and then write them back to the standard output.

Exercise 10.31: Update the program from the previous exercise so that it prints only the unique elements. Your program should use `unqiue_copy` (§ 10.4.1, p. 403).

Exercise 10.32: Rewrite the bookstore problem from § 1.6 (p. 24) using a `vector` to hold the transactions and various algorithms to do the processing. Use `sort` with your `compareIsbn` function from § 10.3.1 (p. 387) to arrange the transactions in order, and then use `find` and `accumulate` to do the sum.

Exercise 10.33: Write a program that takes the names of an input file and two output files. The input file should hold integers. Using an `istream_iterator` read the input file. Using `ostream_iterators`, write the odd numbers into the first output file. Each value should be followed by a space. Write the even numbers into the second file. Each of these values should be placed on a separate line.

10.4.3. Reverse Iterators

A reverse iterator is an iterator that traverses a container backward, from the last element toward the first. A reverse iterator inverts the meaning of increment (and decrement). Incrementing (`++it`) a reverse iterator moves the iterator to the previous element; deremented (`--it`) moves the iterator to the next element.

The containers, aside from `forward_list`, all have reverse iterators. We obtain a reverse iterator by calling the `rbegin`, `rend`, `crbegin`, and `crend` members. These members return reverse iterators to the last element in the container and one “past” (i.e., one before) the beginning of the container. As with ordinary iterators, there are both `const` and nonconst reverse iterators.

Figure 10.1 illustrates the relationship between these four iterators on a hypothetical `vector` named `vec`.

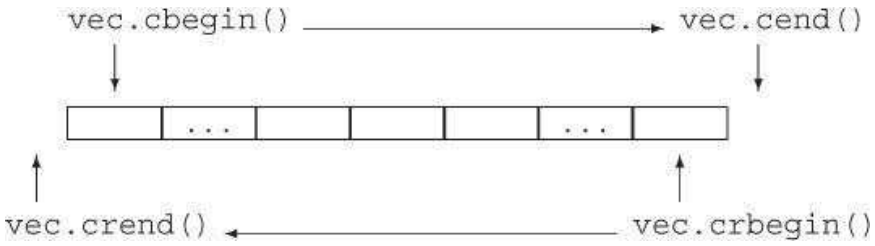


Figure 10.1. Comparing begin/cend and rbegin/crend Iterators

As an example, the following loop prints the elements of `vec` in reverse order:

Click here to view code image

```
vector<int> vec = {0,1,2,3,4,5,6,7,8,9};
// reverse iterator of vector from back to front
for (auto r_iter = vec.crbegin(); // binds r_iter to the last element
          r_iter != vec.crend();  // crend refers 1 before 1st element
           ++r_iter)                // decrements the iterator one
element
     cout << *r_iter << endl;       // prints 9, 8, 7, ... 0
```

Although it may seem confusing to have the meaning of the increment and decrement operators reversed, doing so lets us use the algorithms transparently to process a container forward or backward. For example, we can sort our `vector` in descending order by passing `sort` a pair of reverse iterators:

Click here to view code image

```
sort(vec.begin(), vec.end()); // sorts vec in "normal" order
// sorts in reverse: puts the smallest element at the end of vec
sort(vec.rbegin(), vec.rend());
```

Reverse Iterators Require Decrement Operators

Not surprisingly, we can define a reverse iterator only from an iterator that supports `--` as well as `++`. After all, the purpose of a reverse iterator is to move the iterator backward through the sequence. Aside from `forward_list`, the iterators on the standard containers all support decrement as well as increment. However, the stream iterators do not, because it is not possible to move backward through a stream. Therefore, it is not possible to create a reverse iterator from a `forward_list` or a stream iterator.

Relationship between Reverse Iterators and Other Iterators

Suppose we have a `string` named `line` that contains a comma-separated list of

words, and we want to print the first word in `line`. Using `find`, this task is easy:

Click here to view code image

```
// find the first element in a comma-separated list
auto comma = find(line.cbegin(), line.cend(), ',');
cout << string(line.cbegin(), comma) << endl;
```

If there is a comma in `line`, then `comma` refers to that comma; otherwise it is `line.cend()`. When we print the `string` from `line.cbegin()` to `comma`, we print characters up to the comma, or the entire `string` if there is no comma.

If we wanted the last word, we can use reverse iterators instead:

Click here to view code image

```
// find the last element in a comma-separated list
auto rcomma = find(line.crbegin(), line.crend(), ',');
```

Because we pass `crbegin()` and `crend()`, this call starts with the last character in `line` and searches backward. When `find` completes, if there is a comma, then `rcomma` refers to the last comma in the line—that is, it refers to the first comma found in the backward search. If there is no comma, then `rcomma` is `line.crend()`.

The interesting part comes when we try to print the word we found. The seemingly obvious way

Click here to view code image

```
// WRONG: will generate the word in reverse order
cout << string(line.crbegin(), rcomma) << endl;
```

generates bogus output. For example, had our input been

FIRST,MIDDLE,LAST

then this statement would print TSAL!

Figure 10.2 illustrates the problem: We are using reverse iterators, which process the `string` backward. Therefore, our output statement prints from `crbegin` backward through `line`. Instead, we want to print from `rcomma` forward to the end of `line`. However, we can't use `rcomma` directly. That iterator is a reverse iterator, which means that it goes backward toward the beginning of the `string`. What we need to do is transform `rcomma` back into an ordinary iterator that will go forward through `line`. We can do so by calling the `reverse_iterator`'s `base` member, which gives us its corresponding ordinary iterator:

Click here to view code image

```
// ok: get a forward iterator and read to the end of line
cout << string(rcomma.base(), line.cend()) << endl;
```

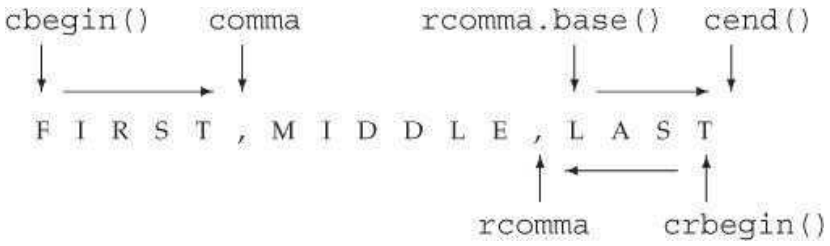


Figure 10.2. Relationship between Reverse and Ordinary Iterators

Given the same preceding input, this statement prints LAST as expected.

The objects shown in Figure 10.2 illustrate the relationship between ordinary and reverse iterators. For example, `rcomma` and `rcomma.base()` refer to different elements, as do `line.crbegin()` and `line.cend()`. These differences are needed to ensure that the *range* of elements, whether processed forward or backward, is the same.

Technically speaking, the relationship between normal and reverse iterators accommodates the properties of a left-inclusive range (§ 9.2.1, p. 331). The point is that `[line.crbegin(), rcomma)` and `[rcomma.base(), line.cend())` refer to the same elements in `line`. In order for that to happen, `rcomma` and `rcomma.base()` must yield adjacent positions, rather than the same position, as must `crbegin()` and `cend()`.

Note

The fact that reverse iterators are intended to represent ranges and that these ranges are asymmetric has an important consequence: When we initialize or assign a reverse iterator from a plain iterator, the resulting iterator does not refer to the same element as the original.

Exercises Section 10.4.3

Exercise 10.34: Use `reverse_iterators` to print a `vector` in reverse order.

Exercise 10.35: Now print the elements in reverse order using ordinary iterators.

Exercise 10.36: Use `find` to find the last element in a `list` of `ints` with value 0.

Exercise 10.37: Given a `vector` that has ten elements, copy the elements from positions 3 through 7 in reverse order to a `list`.

10.5. Structure of Generic Algorithms

The most fundamental property of any algorithm is the list of operations it requires from its iterator(s). Some algorithms, such as `find`, require only the ability to access an element through the iterator, to increment the iterator, and to compare two iterators for equality. Others, such as `sort`, require the ability to read, write, and randomly access elements. The iterator operations required by the algorithms are grouped into five **iterator categories** listed in Table 10.5. Each algorithm specifies what kind of iterator must be supplied for each of its iterator parameters.

Table 10.5. Iterator Categories

Input iterator	Read, but not write; single-pass, increment only
Output iterator	Write, but not read; single-pass, increment only
Forward iterator	Read and write; multi-pass, increment only
Bidirectional iterator	Read and write; multi-pass, increment and decrement
Random-access iterator	Read and write; multi-pass, full iterator arithmetic

A second way is to classify the algorithms (as we did in the beginning of this chapter) is by whether they read, write, or reorder the elements in the sequence. Appendix A covers all the algorithms according to this classification.

The algorithms also share a set of parameter-passing conventions and a set of naming conventions, which we shall cover after looking at iterator categories.

10.5.1. The Five Iterator Categories

Like the containers, iterators define a common set of operations. Some operations are provided by all iterators; other operations are supported by only specific kinds of iterators. For example, `ostream_iterators` have only increment, dereference, and assignment. Iterators on `vector`, `strings`, and `deques` support these operations and the decrement, relational, and arithmetic operators.

Iterators are categorized by the operations they provide and the categories form a sort of hierarchy. With the exception of output iterators, an iterator of a higher category provides all the operations of the iterators of a lower categories.

The standard specifies the minimum category for each iterator parameter of the generic and numeric algorithms. For example, `find`—which implements a one-pass, read-only traversal over a sequence—minimally requires an input iterator. The `replace` function requires a pair of iterators that are at least forward iterators. Similarly, `replace_copy` requires forward iterators for its first two iterators. Its third iterator, which represents a destination, must be at least an output iterator, and so

on. For each parameter, the iterator must be at least as powerful as the stipulated minimum. Passing an iterator of a lesser power is an error.

> **Warning**
>
> Many compilers will not complain when we pass the wrong category of iterator to an algorithm.

The Iterator Categories

Input iterators: can read elements in a sequence. An input iterator must provide

- Equality and inequality operators (`==`, `!=`) to compare two iterators
- Prefix and postfix increment (`++`) to advance the iterator
- Dereference operator (`*`) to read an element; dereference may appear only on the right-hand side of an assignment
- The arrow operator (`->`) as a synonym for `(* it).member`—that is, dereference the iterator and fetch a member from the underlying object

Input iterators may be used only sequentially. We are guaranteed that `*it++` is valid, but incrementing an input iterator may invalidate all other iterators into the stream. As a result, there is no guarantee that we can save the state of an input iterator and examine an element through that saved iterator. Input iterators, therefore, may be used only for single-pass algorithms. The `find` and `accumulate` algorithms require input iterators; `istream_iterators` are input iterators.

Output iterators: can be thought of as having complementary functionality to input iterators; they write rather than read elements. Output iterators must provide

- Prefix and postfix increment (`++`) to advance the iterator
- Dereference (`*`), which may appear only as the left-hand side of an assignment (Assigning to a dereferenced output iterator writes to the underlying element.)

We may assign to a given value of an output iterator only once. Like input iterators, output iterators may be used only for single-pass algorithms. Iterators used as a destination are typically output iterators. For example, the third parameter to `copy` is an output iterator. The `ostream_iterator` type is an output iterator.

Forward iterators: can read and write a given sequence. They move in only one direction through the sequence. Forward iterators support all the operations of both input iterators and output iterators. Moreover, they can read or write the same element multiple times. Therefore, we can use the saved state of a forward iterator. Hence, algorithms that use forward iterators may make multiple passes through the sequence. The `replace` algorithm requires a forward iterator; iterators on `forward_list` are forward iterators.

Bidirectional iterators: can read and write a sequence forward or backward. In addition to supporting all the operations of a forward iterator, a bidirectional iterator also supports the prefix and postfix decrement (`--`) operators. The `reverse` algorithm requires bidirectional iterators, and aside from `forward_list`, the library containers supply iterators that meet the requirements for a bidirectional iterator.

Random-access iterators: provide constant-time access to any position in the sequence. These iterators support all the functionality of bidirectional iterators. In addition, random-access iterators support the operations from Table 3.7 (p. 111):

- The relational operators (<, <=, >, and >=) to compare the relative positions of two iterators.
- Addition and subtraction operators (`+`, `+=`, `-`, and `-=`) on an iterator and an integral value. The result is the iterator advanced (or retreated) the integral number of elements within the sequence.
- The subtraction operator (`-`) when applied to two iterators, which yields the distance between two iterators.
- The subscript operator (`iter[n]`) as a synonym for `* (iter + n)`.

The `sort` algorithms require random-access iterators. Iterators for `array`, `deque`, `string`, and `vector` are random-access iterators, as are pointers when used to access elements of a built-in array.

Exercises Section 10.5.1

Exercise 10.38: List the five iterator categories and the operations that each supports.

Exercise 10.39: What kind of iterator does a `list` have? What about a `vector`?

Exercise 10.40: What kinds of iterators do you think `copy` requires? What about `reverse` or `unique`?

10.5.2. Algorithm Parameter Patterns

Superimposed on any other classification of the algorithms is a set of parameter conventions. Understanding these parameter conventions can aid in learning new algorithms—by knowing what the parameters mean, you can concentrate on understanding the operation the algorithm performs. Most of the algorithms have one of the following four forms:

Click here to view code image

```
alg(beg, end, other args);
```

```
alg(beg, end, dest, other args);
alg(beg, end, beg2, other args);
alg(beg, end, beg2, end2, other args);
```

where *alg* is the name of the algorithm, and `beg` and `end` denote the input range on which the algorithm operates. Although nearly all algorithms take an input range, the presence of the other parameters depends on the work being performed. The common ones listed here—`dest`, `beg2`, and `end2`—are all iterators. When used, these iterators fill similar roles. In addition to these iterator parameters, some algorithms take additional, noniterator parameters that are algorithm specific.

Algorithms with a Single Destination Iterator

A `dest` parameter is an iterator that denotes a destination in which the algorithm can write its output. Algorithms *assume* that it is safe to write as many elements as needed.

> **Warning**
>
> Algorithms that write to an output iterator assume the destination is large enough to hold the output.

If `dest` is an iterator that refers directly to a container, then the algorithm writes its output to existing elements within the container. More commonly, `dest` is bound to an insert iterator (§ 10.4.1, p. 401) or an `ostream_iterator` (§ 10.4.2, p. 403). An insert iterator adds new elements to the container, thereby ensuring that there is enough space. An `ostream_iterator` writes to an output stream, again presenting no problem regardless of how many elements are written.

Algorithms with a Second Input Sequence

Algorithms that take either `beg2` alone or `beg2` and `end2` use those iterators to denote a second input range. These algorithms typically use the elements from the second range in combination with the input range to perform a computation.

When an algorithm takes both `beg2` and `end2`, these iterators denote a second range. Such algorithms take two completely specified ranges: the input range denoted by `[beg, end)`, and a second input range denoted by `[beg2, end2)`.

Algorithms that take only `beg2` (and not `end2`) treat `beg2` as the first element in a second input range. The end of this range is not specified. Instead, these algorithms *assume* that the range starting at `beg2` is at least as large as the one denoted by `beg, end`.

Warning

Algorithms that take `beg2` alone *assume* that the sequence beginning at `beg2` is as large as the range denoted by `beg` and `end`.

10.5.3. Algorithm Naming Conventions

Separate from the parameter conventions, the algorithms also conform to a set of naming and overload conventions. These conventions deal with how we supply an operation to use in place of the default < or == operator and with whether the algorithm writes to its input sequence or to a separate destination.

Some Algorithms Use Overloading to Pass a Predicate

Algorithms that take a predicate to use in place of the < or == operator, and that do not take other arguments, typically are overloaded. One version of the function uses the element type's operator to compare elements; the second takes an extra parameter that is a predicate to use in place of < or ==:

Click here to view code image

```
unique(beg, end);        // uses the == operator to compare the elements
unique(beg, end, comp); // uses comp to compare the elements
```

Both calls reorder the given sequence by removing adjacent duplicated elements. The first uses the element type's == operator to check for duplicates; the second calls `comp` to decide whether two elements are equal. Because the two versions of the function differ as to the number of arguments, there is no possible ambiguity (§ 6.4, p. 233) as to which function is being called.

Algorithms with _if Versions

Algorithms that take an element value typically have a second named (not overloaded) version that takes a predicate (§ 10.3.1, p. 386) in place of the value. The algorithms that take a predicate have the suffix `_if` appended:

Click here to view code image

```
find(beg, end, val);      // find the first instance of val in the input range
find_if(beg, end, pred); // find the first instance for which pred is true
```

These algorithms both find the first instance of a specific element in the input range. The `find` algorithm looks for a specific value; the `find_if` algorithm looks for a value for which `pred` returns a nonzero value.

These algorithms provide a named version rather than an overloaded one because both versions of the algorithm take the same number of arguments. Overloading ambiguities would therefore be possible, albeit rare. To avoid any possible ambiguities, the library provides separate named versions for these algorithms.

Distinguishing Versions That Copy from Those That Do Not

By default, algorithms that rearrange elements write the rearranged elements back into the given input range. These algorithms provide a second version that writes to a specified output destination. As we've seen, algorithms that write to a destination append `_copy` to their names (§ 10.2.2, p. 383):

Click here to view code image

```
reverse(beg, end);              // reverse the elements in the input range
reverse_copy(beg, end, dest);// copy elements in reverse order into dest
```

Some algorithms provide both `_copy` and `_if` versions. These versions take a destination iterator and a predicate:

Click here to view code image

```
// removes the odd elements from v1
remove_if(v1.begin(), v1.end(),
                      [](int i) { return i % 2; });
// copies only the even elements from v1 into v2; v1 is unchanged
remove_copy_if(v1.begin(), v1.end(), back_inserter(v2),
                [](int i) { return i % 2; });
```

Both calls use a lambda (§ 10.3.2, p. 388) to determine whether an element is odd. In the first case, we remove the odd elements from the input sequence itself. In the second, we copy the non-odd (aka even) elements from the input range into `v2`.

Exercises Section 10.5.3

Exercise 10.41: Based only on the algorithm and argument names, describe the operation that the each of the following library algorithms performs:

Click here to view code image

```
replace(beg, end, old_val, new_val);
replace_if(beg, end, pred, new_val);
replace_copy(beg, end, dest, old_val, new_val);
replace_copy_if(beg, end, dest, pred, new_val);
```

10.6. Container-Specific Algorithms

Unlike the other containers, `list` and `forward_list` define several algorithms as members. In particular, the list types define their own versions of `sort, merge, remove, reverse`, and `unique`. The generic version of `sort` requires random-access iterators. As a result, `sort` cannot be used with `list` and `forward_list` because these types offer bidirectional and forward iterators, respectively.

The generic versions of the other algorithms that the list types define can be used with lists, but at a cost in performance. These algorithms swap elements in the input sequence. A list can "swap" its elements by changing the links among its elements rather than swapping the values of those elements. As a result, the list-specific versions of these algorithms can achieve much better performance than the corresponding generic versions.

These `list`-specific operations are described in Table 10.6. Generic algorithms not listed in the table that take appropriate iterators execute equally efficiently on `lists` and `forward_listss` as on other containers.

Table 10.6. Algorithms That are Members of list and forward_list

	These operations return `void`.
`lst.merge(lst2)` `lst.merge(lst2, comp)`	Merges elements from `lst2` onto `lst`. Both `lst` and `lst2` must be sorted. Elements are removed from `lst2`. After the `merge`, `lst2` is empty. The first version uses the `<` operator; the second version uses the given comparison operation.
`lst.remove(val)` `lst.remove_if(pred)`	Calls `erase` to remove each element that is `==` to the given value or for which the given unary predicate succeeds.
`lst.reverse()`	Reverses the order of the elements in `lst`.
`lst.sort()` `lst.sort(comp)`	Sorts the elements of `lst` using `<` or the given comparison operation.
`lst.unique()` `lst.unique(pred)`	Calls `erase` to remove consecutive copies of the same value. The first version uses `==`; the second uses the given binary predicate.

Best Practices

The list member versions should be used in preference to the generic algorithms for `lists` and `forward_lists`.

The splice Members

The list types also define a `splice` algorithm, which is described in Table 10.7. This algorithm is particular to list data structures. Hence a generic version of this algorithm is not needed.

Table 10.7. Arguments to the list and forward_list splice Members

	`lst.splice(`*args*`)` or `flst.splice_after(`*args*`)`
`(p, lst2)`	`p` is an iterator to an element in `lst` or an iterator just before an element in `flst`. Moves all the element(s) from `lst2` into `lst` just before `p` or into `flst` just after `p`. Removes the element(s) from `lst2`. `lst2` must have the same type as `lst` or `flst` and may not be the same list.
`(p, lst2, p2)`	`p2` is a valid iterator into `lst2`. Moves the element denoted by `p2` into `lst` or moves the element just after `p2` into `flst`. `lst2` can be the same list as `lst` or `flst`.
`(p, lst2, b, e)`	`b` and `e` must denote a valid range in `lst2`. Moves the elements in the given range from `lst2`. `lst2` and `lst` (or `flst`) can be the same list but `p` must not denote an element in the given range.

The List-Specific Operations Do Change the Containers

Most of the list-specific algorithms are similar—but not identical—to their generic counterparts. However, a crucially important difference between the list-specific and the generic versions is that the list versions change the underlying container. For example, the list version of `remove` removes the indicated elements. The list version of `unique` removes the second and subsequent duplicate elements.

Similarly, `merge` and `splice` are destructive on their arguments. For example, the generic version of `merge` writes the merged sequence to a given destination iterator; the two input sequences are unchanged. The list `merge` function destroys the given list—elements are removed from the argument list as they are merged into the object on which `merge` was called. After a `merge`, the elements from both lists continue to exist, but they are all elements of the same list.

Exercises Section 10.6

Exercise 10.42: Reimplement the program that eliminated duplicate words that we wrote in § 10.2.3 (p. 383) to use a `list` instead of a `vector`.

Chapter Summary

The standard library defines about 100 type-independent algorithms that operate on sequences. Sequences can be elements in a library container type, a built-in array, or generated (for example) by reading or writing to a stream. Algorithms achieve their type independence by operating in terms of iterators. Most algorithms take as their first two arguments a pair of iterators denoting a range of elements. Additional iterator arguments might include an output iterator denoting a destination, or another

iterator or iterator pair denoting a second input sequence.

Iterators are categorized into one of five categories depending on the operations they support. The iterator categories are input, output, forward, bidirectional, and random access. An iterator belongs to a particular category if it supports the operations required for that iterator category.

Just as iterators are categorized by their operations, iterator parameters to the algorithms are categorized by the iterator operations they require. Algorithms that only read their sequences require only input iterator operations. Those that write to a destination iterator require only the actions of an output iterator, and so on.

Algorithms *never* directly change the size of the sequences on which they operate. They may copy elements from one position to another but cannot directly add or remove elements.

Although algorithms cannot add elements to a sequence, an insert iterator may do so. An insert iterator is bound to a container. When we assign a value of the container's element type to an insert iterator, the iterator adds the given element to the container.

The `forward_list` and `list` containers define their own versions of some of the generic algorithms. Unlike the generic algorithms, these list-specific versions modify the given lists.

Defined Terms

back_inserter Iterator adaptor that takes a reference to a container and generates an insert iterator that uses `push_back` to add elements to the specified container.

bidirectional iterator Same operations as forward iterators plus the ability to use `--` to move backward through the sequence.

binary predicate Predicate that has two parameters.

bind Library function that binds one or more arguments to a callable expression. `bind` is defined in the `functional` header.

callable object Object that can appear as the left-hand operand of the call operator. Pointers to functions, lambdas, and objects of a class that defines an overloaded function call operator are all callable objects.

capture list Portion of a lambda expression that specifies which variables from the surrounding context the lambda expression may access.

cref Library function that returns a copyable object that holds a reference to a `const` object of a type that cannot be copied.

forward iterator Iterator that can read and write elements but is not required to support `--`.

front_inserter Iterator adaptor that, given a container, generates an insert iterator that uses `push_front` to add elements to the beginning of that container.

generic algorithms Type-independent algorithms.

input iterator Iterator that can read, but not write, elements of a sequence.

insert iterator Iterator adaptor that generates an iterator that uses a container operation to add elements to a given container.

inserter Iterator adaptor that takes an iterator and a reference to a container and generates an insert iterator that uses `insert` to add elements just ahead of the element referred to by the given iterator.

istream_iterator Stream iterator that reads an input stream.

iterator categories Conceptual organization of iterators based on the operations that an iterator supports. Iterator categories form a hierarchy, in which the more powerful categories offer the same operations as the lesser categories. The algorithms use iterator categories to specify what operations the iterator arguments must support. As long as the iterator provides at least that level of operation, it can be used. For example, some algorithms require only input iterators. Such algorithms can be called on any iterator other than one that meets only the output iterator requirements. Algorithms that require random-access iterators can be used only on iterators that support random-access operations.

lambda expression Callable unit of code. A lambda is somewhat like an unnamed, inline function. A lambda starts with a capture list, which allows the lambda to access variables in the enclosing function. Like a function, it has a (possibly empty) parameter list, a return type, and a function body. A lambda can omit the return type. If the function body is a single `return` statement, the return type is inferred from the type of the object that is returned. Otherwise, an omitted return type defaults to `void`.

move iterator Iterator adaptor that generates an iterator that moves elements instead of copying them. Move iterators are covered in Chapter 13.

ostream_iterator Iterator that writes to an output stream.

output iterator Iterator that can write, but not necessarily read, elements.

predicate Function that returns a type that can be converted to `bool`. Often used by the generic algorithms to test elements. Predicates used by the library are either unary (taking one argument) or binary (taking two).

random-access iterator Same operations as bidirectional iterators plus the

relational operators to compare iterator values, and the subscript operator and arithmetic operations on iterators, thus supporting random access to elements.

ref Library function that generates a copyable object from a reference to an object of a type that cannot be copied.

reverse iterator Iterator that moves backward through a sequence. These iterators exchange the meaning of ++ and --.

stream iterator Iterator that can be bound to a stream.

unary predicate Predicate that has one parameter.

Chapter 11. Associative Containers

Contents

Associative and sequential containers differ from one another in a fundamental way: Elements in an associative container are stored and retrieved by a key. In contrast, elements in a sequential container are stored and accessed sequentially by their position in the container.

Although the associative containers share much of the behavior of the sequential containers, they differ from the sequential containers in ways that reflect the use of keys.

Associative containers support efficient lookup and retrieval by a key. The two primary **associative-container** types are **map** and **set**. The elements in a `map` are key–value pairs: The key serves as an index into the `map`, and the value represents the data associated with that index. A `set` element contains only a key; a `set` supports efficient queries as to whether a given key is present. We might use a `set` to hold words that we want to ignore during some kind of text processing. A dictionary would be a good use for a `map`: The word would be the key, and its definition would be the value.

The library provides eight associative containers, listed in Table 11.1. These eight differ along three dimensions: Each container is (1) a `set` or a `map`, (2) requires unique keys or allows multiple keys, and (3) stores the elements in order or not. The

containers that allow multiple keys include the word `multi`; those that do not keep their keys ordered start with the word `unordered`. Hence an `unordered_multi_set` is a set that allows multiple keys whose elements are not stored in order, whereas a `set` has unique keys that are stored in order. The unordered containers use a hash function to organize their elements. We'll have more to say about the hash function in § 11.4 (p. 444).

Table 11.1. Associative Container Types

Elements Ordered by Key	
`map`	Associative array; holds key–value pairs
`set`	Container in which the key is the value
`multimap`	`map` in which a key can appear multiple times
`multiset`	`set` in which a key can appear multiple times
Unordered Collections	
`unordered_map`	`map` organized by a hash function
`unordered_set`	`set` organized by a hash function
`unordered_multimap`	Hashed `map`; keys can appear multiple times
`unordered_multiset`	Hashed `set`; keys can appear multiple times

The `map` and **multimap** types are defined in the `map` header; the `set` and **multiset** types are in the `set` header; and the unordered containers are in the `unordered_map` and `unordered_set` headers.

11.1. Using an Associative Container

Although most programmers are familiar with data structures such as `vectors` and `lists`, many have never used an associative data structure. Before we look at the details of how the library supports these types, it will be helpful to start with examples of how we can use these containers.

A `map` is a collection of key–value pairs. For example, each pair might contain a person's name as a key and a phone number as its value. We speak of such a data structure as "mapping names to phone numbers." The `map` type is often referred to as an **associative array**. An associative array is like a "normal" array except that its subscripts don't have to be integers. Values in a `map` are found by a key rather than by their position. Given a `map` of names to phone numbers, we'd use a person's name as a subscript to fetch that person's phone number.

In contrast, a `set` is simply a collection of keys. A `set` is most useful when we simply want to know whether a value is present. For example, a business might define a `set` named `bad_checks` to hold the names of individuals who have written bad checks. Before accepting a check, that business would query `bad_checks` to see whether the customer's name was present.

Using a map

A classic example that relies on associative arrays is a word-counting program:

Click here to view code image

```
// count the number of times each word occurs in the input
map<string, size_t> word_count; // empty map from string to size_t
string word;
while (cin >> word)
        ++word_count[word];    // fetch and increment the counter for word
for (const auto &w : word_count) // for each element in the map
    // print the results
    cout <<  w.first << " occurs " << w.second
         << ((w.second > 1) ? " times" : " time") << endl;
```

This program reads its input and reports how often each word appears.

Like the sequential containers, the associative containers are templates (§ 3.3, p. 96). To define a `map`, we must specify both the key and value types. In this program, the `map` stores elements in which the keys are `strings` and the values are `size_ts` (§ 3.5.2, p. 116). When we subscript `word_count`, we use a `string` as the subscript, and we get back the `size_t` counter associated with that `string`.

The `while` loop reads the standard input one word at a time. It uses each word to subscript `word_count`. If `word` is not already in the `map`, the subscript operator creates a new element whose key is `word` and whose value is `0`. Regardless of whether the element had to be created, we increment the value.

Once we've read all the input, the range `for` (§ 3.2.3, p. 91) iterates through the `map`, printing each word and the corresponding counter. When we fetch an element from a `map`, we get an object of type `pair`, which we'll describe in § 11.2.3 (p. 426). Briefly, a `pair` is a template type that holds two (`public`) data elements named `first` and `second`. The `pairs` used by `map` have a `first` member that is the key and a `second` member that is the corresponding value. Thus, the effect of the output statement is to print each word and its associated counter.

If we ran this program on the text of the first paragraph in this section, our output would be

```
Although occurs 1 time
Before occurs 1 time
an occurs 1 time
and occurs 1 time
...
```

Using a set

A logical extension to our program is to ignore common words like “the,” “and,” “or,” and so on. We’ll use a `set` to hold the words we want to ignore and count only those words that are not in this set:

Click here to view code image

```
// count the number of times each word occurs in the input
map<string, size_t> word_count; // empty map from string to size_t
set<string> exclude = {"The", "But", "And", "Or", "An", "A",
                       "the", "but", "and", "or", "an",
"a"};
string word;
while (cin >> word)
    // count only words that are not in exclude
    if (exclude.find(word) == exclude.end())
        ++word_count[word];    // fetch and increment the counter for word
```

Like the other containers, `set` is a template. To define a `set`, we specify the type of its elements, which in this case are `strings`. As with the sequential containers, we can list initialize (§ 9.2.4, p. 336) the elements of an associative container. Our `exclude` set holds the 12 words we want to ignore.

The important difference between this program and the previous program is that before counting each word, we check whether the word is in the exclusion set. We do this check in the `if`:

Click here to view code image

```
// count only words that are not in exclude
if (exclude.find(word) == exclude.end())
```

The call to `find` returns an iterator. If the given key is in the `set`, the iterator refers to that key. If the element is not found, `find` returns the off-the-end iterator. In this version, we update the counter for `word` only if `word` is not in `exclude`.

If we run this version on the same input as before, our output would be

```
Although occurs 1 time
Before occurs 1 time
are occurs 1 time
as occurs 1 time
...
```

Exercises Section 11.1

Exercise 11.1: Describe the differences between a `map` and a `vector`.

Exercise 11.2: Give an example of when each of `list`, `vector`, `deque`, `map`, and `set` might be most useful.

Exercise 11.3: Write your own version of the word-counting program.

Exercise 11.4: Extend your program to ignore case and punctuation. For example, “example.” “example,” and “Example” should all increment the same counter.

11.2. Overview of the Associative Containers

Associative containers (both ordered and unordered) support the general container operations covered in § 9.2 (p. 328) and listed in Table 9.2 (p. 330). The associative containers do *not* support the sequential-container position-specific operations, such as `push_front` or `back`. Because the elements are stored based on their keys, these operations would be meaningless for the associative containers. Moreover, the associative containers do not support the constructors or insert operations that take an element value and a count.

In addition to the operations they share with the sequential containers, the associative containers provide some operations (Table 11.7 (p. 438)) and type aliases (Table 11.3 (p. 429)) that the sequential containers do not. In addition, the unordered containers provide operations for tuning their hash performance, which we'll cover in § 11.4 (p. 444).

The associative container iterators are bidirectional (§ 10.5.1, p. 410).

11.2.1. Defining an Associative Container

As we've just seen, when we define a `map`, we must indicate both the key and value type; when we define a `set`, we specify only a key type, because there is no value type. Each of the associative containers defines a default constructor, which creates an empty container of the specified type. We can also initialize an associative container as a copy of another container of the same type or from a range of values, so long as those values can be converted to the type of the container. Under the new standard, we can also list initialize the elements:

Click here to view code image

```
map<string, size_t> word_count;  // empty
// list initialization
set<string> exclude = {"the", "but", "and", "or", "an", "a",
                       "The", "But", "And", "Or", "An",
"A"};
// three elements; authors maps last name to first
map<string, string> authors = { {"Joyce", "James"},
                                {"Austen", "Jane"},
```

```
                                        {"Dickens", "Charles"} };
```

As usual, the initializers must be convertible to the type in the container. For `set`, the element type is the key type.

When we initialize a `map`, we have to supply both the key and the value. We wrap each key–value pair inside curly braces:

```
{key, value}
```

to indicate that the items together form one element in the `map`. The key is the first element in each pair, and the value is the second. Thus, `authors` maps last names to first names, and is initialized with three elements.

Initializing a multimap or multiset

The keys in a `map` or a `set` must be unique; there can be only one element with a given key. The `multimap` and `multiset` containers have no such restriction; there can be several elements with the same key. For example, the `map` we used to count words must have only one element per given word. On the other hand, a dictionary could have several definitions associated with a particular word.

The following example illustrates the differences between the containers with unique keys and those that have multiple keys. First, we'll create a `vector` of `ints` named `ivec` that has 20 elements: two copies of each of the integers from 0 through 9 inclusive. We'll use that `vector` to initialize a `set` and a `multiset`:

Click here to view code image

```
// define a vector with 20 elements, holding two copies of each number from 0 to 9
vector<int> ivec;
for (vector<int>::size_type i = 0; i != 10; ++i) {
    ivec.push_back(i);
    ivec.push_back(i);  // duplicate copies of each number
}
// iset holds unique elements from ivec; miset holds all 20 elements
set<int> iset(ivec.cbegin(), ivec.cend());
multiset<int> miset(ivec.cbegin(), ivec.cend());
cout << ivec.size() << endl;     // prints 20
cout << iset.size() << endl;     // prints 10
cout << miset.size() << endl;    // prints 20
```

Even though we initialized `iset` from the entire `ivec` container, `iset` has only ten elements: one for each distinct element in `ivec`. On the other hand, `miset` has 20 elements, the same as the number of elements in `ivec`.

Exercises Section 11.2.1

Exercise 11.5: Explain the difference between a `map` and a `set`. When

might you use one or the other?

Exercise 11.6: Explain the difference between a `set` and a `list`. When might you use one or the other?

Exercise 11.7: Define a `map` for which the key is the family's last name and the value is a `vector` of the children's names. Write code to add new families and to add new children to an existing family.

Exercise 11.8: Write a program that stores the excluded words in a `vector` instead of in a `set`. What are the advantages to using a `set`?

11.2.2. Requirements on Key Type

The associative containers place constraints on the type that is used as a key. We'll cover the requirements for keys in the unordered containers in § 11.4 (p. 445). For the ordered containers—`map`, `multimap`, `set`, and `multiset`—the key type must define a way to compare the elements. By default, the library uses the < operator for the key type to compare the keys. In the set types, the key is the element type; in the map types, the key is the first type. Thus, the key type for `word_count` in § 11.1 (p. 421) is `string`. Similarly, the key type for `exclude` is `string`.

Note

Callable objects passed to a sort algorithm (§ 10.3.1, p. 386) must meet the same requirements as do the keys in an associative container.

Key Types for Ordered Containers

Just as we can provide our own comparison operation to an algorithm (§ 10.3, p. 385), we can also supply our own operation to use in place of the < operator on keys. The specified operation must define a **strict weak ordering** over the key type. We can think of a strict weak ordering as "less than," although our function might use a more complicated procedure. However we define it, the comparison function must have the following properties:

- Two keys cannot both be "less than" each other; if `k1` is "less than" `k2`, then `k2` must never be "less than" `k1`.
- If `k1` is "less than" `k2` and `k2` is "less than" `k3`, then `k1` must be "less than" `k3`.
- If there are two keys, and neither key is "less than" the other, then we'll say that those keys are "equivalent." If `k1` is "equivalent" to `k2` and `k2` is

"equivalent" to k3, then k1 must be "equivalent" to k3.

If two keys are equivalent (i.e., if neither is "less than" the other), the container treats them as equal. When used as a key to a map, there will be only one element associated with those keys, and either key can be used to access the corresponding value.

Note

In practice, what's important is that a type that defines a < operator that "behaves normally" can be used as a key.

Using a Comparison Function for the Key Type

The type of the operation that a container uses to organize its elements is part of the type of that container. To specify our own operation, we must supply the type of that operation when we define the type of an associative container. The operation type is specified following the element type inside the angle brackets that we use to say which type of container we are defining.

Each type inside the angle brackets is just that, a type. We supply a particular comparison operation (that must have the same type as we specified inside the angle brackets) as a constructor argument when we create a container.

For example, we can't directly define a multiset of Sales_data because Sales_data doesn't have a < operator. However, we can use the compareIsbn function from the exercises in § 10.3.1 (p. 387) to define a multiset. That function defines a strict weak ordering based on their ISBNs of two given Sales_data objects. The compareIsbn function should look something like

Click here to view code image

```
bool compareIsbn(const Sales_data &lhs, const Sales_data
&rhs)
{
    return lhs.isbn() < rhs.isbn();
}
```

To use our own operation, we must define the multiset with two types: the key type, Sales_data, and the comparison type, which is a function pointer type (§ 6.7, p. 247) that can point to compareIsbn. When we define objects of this type, we supply a pointer to the operation we intend to use. In this case, we supply a pointer to compareIsbn:

Click here to view code image

```
// bookstore can have several transactions with the same ISBN
// elements in bookstore will be in ISBN order
```

```
multiset<Sales_data, decltype(compareIsbn)*>
    bookstore(compareIsbn);
```

Here, we use `decltype` to specify the type of our operation, remembering that when we use `decltype` to form a function pointer, we must add a `*` to indicate that we're using a pointer to the given function type (§ 6.7, p. 250). We initialize `bookstore` from `compareIsbn`, which means that when we add elements to `bookstore`, those elements will be ordered by calling `compareIsbn`. That is, the elements in `bookstore` will be ordered by their `ISBN` members. We can write `compareIsbn` instead of `&compareIsbn` as the constructor argument because when we use the name of a function, it is automatically converted into a pointer if needed (§ 6.7, p. 248). We could have written `&compareIsbn` with the same effect.

Exercises Section 11.2.2

Exercise 11.9: Define a `map` that associates words with a `list` of line numbers on which the word might occur.

Exercise 11.10: Could we define a `map` from `vector<int>::iterator` to `int`? What about from `list<int>::iterator` to `int`? In each case, if not, why not?

Exercise 11.11: Redefine `bookstore` without using `decltype`.

11.2.3. The `pair` Type

Before we look at the operations on associative containers, we need to know about the library type named **pair**, which is defined in the `utility` header.

A `pair` holds two data members. Like the containers, `pair` is a template from which we generate specific types. We must supply two type names when we create a `pair`. The data members of the `pair` have the corresponding types. There is no requirement that the two types be the same:

Click here to view code image

```
pair<string, string> anon;        // holds two strings
pair<string, size_t> word_count; // holds a string and an size_t
pair<string, vector<int>> line;  // holds string and vector<int>
```

The default `pair` constructor value initializes (§ 3.3.1, p. 98) the data members. Thus, `anon` is a `pair` of two empty `strings`, and `line` holds an empty `string` and an empty `vector`. The `size_t` value in `word_count` gets the value 0, and the `string` member is initialized to the empty `string`.

We can also provide initializers for each member:

Click here to view code image

```
pair<string, string> author{"James", "Joyce"};
```

creates a `pair` named `author`, initialized with the values `"James"` and `"Joyce"`.

Unlike other library types, the data members of `pair` are `public` (§ 7.2, p. 268). These members are named `first` and `second`, respectively. We access these members using the normal member access notation (§ 1.5.2, p. 23), as, for example, we did in the output statement of our word-counting program on page 421:

Click here to view code image

```
// print the results
cout <<  w.first << " occurs " << w.second
     << ((w.second > 1) ? " times" : " time") << endl;
```

Here, `w` is a reference to an element in a `map`. Elements in a `map` are `pairs`. In this statement we print the `first` member of the element, which is the key, followed by the `second` member, which is the counter. The library defines only a limited number of operations on `pairs`, which are listed in Table 11.2.

Table 11.2. Operations on pairs

`pair<T1, T2> p;`	`p` is a `pair` with value initialized (§ 3.3.1, p. 98) members of types `T1` and `T2`, respectively.
`pair<T1, T2> p(v1, v2);`	`p` is a `pair` with types `T1` and `T2`; the `first` and `second` members are initialized from `v1` and `v2`, respectively.
`pair<T1, T2> p = {v1, v2};`	Equivalent to `p(v1, v2)`.
`make_pair(v1, v2)`	Returns a `pair` initialized from `v1` and `v2`. The type of the `pair` is inferred from the types of `v1` and `v2`.
`p.first`	Returns the (`public`) data member of `p` named `first`.
`p.second`	Returns the (`public`) data member of `p` named `second`.
`p1` *relop* `p2`	Relational operators (`<`, `>`, `<=`, `>=`). Relational operators are defined as dictionary ordering: For example, `p1 < p2` is `true` if `p1.first < p2.first` or if `!(p2.first < p1.first) && p1.second < p2.second`. Uses the element's `<` operator.
`p1 == p2` `p1 != p2`	Two `pairs` are equal if their `first` and `second` members are respectively equal. Uses the element's `==` operator.

A Function to Create pair Objects

Imagine we have a function that needs to return a `pair`. Under the new standard we can list initialize the return value (§ 6.3.2, p. 226):

C++ 11

Click here to view code image

```
pair<string, int>
process(vector<string> &v)
{
      // process v
      if (!v.empty())
          return {v.back(), v.back().size()}; // list initialize
      else
           return pair<string, int>(); // explicitly constructed return
value
}
```

If `v` isn't empty, we return a `pair` composed of the last `string` in `v` and the size of that `string`. Otherwise, we explicitly construct and return an empty `pair`.

Under earlier versions of C++, we couldn't use braced initializers to return a type like `pair`. Instead, we might have written both returns to explicitly construct the return value:

Click here to view code image

```
if (!v.empty())
    return pair<string, int>(v.back(), v.back().size());
```

Alternatively, we could have used `make_pair` to generate a new `pair` of the appropriate type from its two arguments:

Click here to view code image

```
if (!v.empty())
      return make_pair(v.back(), v.back().size());
```

Exercises Section 11.2.3

Exercise 11.12: Write a program to read a sequence of `strings` and `ints`, storing each into a `pair`. Store the `pairs` in a `vector`.

Exercise 11.13: There are at least three ways to create the `pairs` in the program for the previous exercise. Write three versions of that program, creating the `pairs` in each way. Explain which form you think is easiest to write and understand, and why.

Exercise 11.14: Extend the `map` of children to their family name that you wrote for the exercises in § 11.2.1 (p. 424) by having the `vector` store a `pair` that holds a child's name and birthday.

11.3. Operations on Associative Containers

In addition to the types listed in Table 9.2 (p. 330), the associative containers define the types listed in Table 11.3. These types represent the container's key and value

types.

Table 11.3. Associative Container Additional Type Aliases

`key_type`	Type of the key for this container type
`mapped_type`	Type associated with each key; **`map` types only**
`value_type`	For `set`s, same as the `key_type` For `map`s, `pair<const key_type, mapped_type>`

For the `set` types, the **key_type** and the **value_type** are the same; the values held in a `set` are the keys. In a `map`, the elements are key–value pairs. That is, each element is a `pair` object containing a key and a associated value. Because we cannot change an element's key, the key part of these `pairs` is `const`:

Click here to view code image

```
set<string>::value_type v1;       // v1 is a string
set<string>::key_type v2;         // v2 is a string
map<string, int>::value_type v3; // v3 is a pair<const string, int>
map<string, int>::key_type v4;    // v4 is a string
map<string, int>::mapped_type v5; // v5 is an int
```

As with the sequential containers (§ 9.2.2, p. 332), we use the scope operator to fetch a type member—for example, `map<string, int>::key_type`.

Only the `map` types (`unordered_map`, `unordered_multimap`, `multimap`, and `map`) define **mapped_type**.

11.3.1. Associative Container Iterators

When we dereference an iterator, we get a reference to a value of the container's `value_type`. In the case of `map`, the `value_type` is a `pair` in which `first` holds the `const` key and `second` holds the value:

Click here to view code image

```
// get an iterator to an element in word_count
auto map_it = word_count.begin();
// *map_it is a reference to a pair<const string, size_t> object
cout << map_it->first;           // prints the key for this element
cout << " " << map_it->second;   // prints the value of the element
map_it->first = "new key";       // error: key is const
++map_it->second;      // ok: we can change the value through an iterator
```

Note

It is essential to remember that the `value_type` of a `map` is a `pair` and

that we can change the value but not the key member of that `pair`.

Iterators for sets Are const

Although the `set` types define both the `iterator` and `const_iterator` types, both types of iterators give us read-only access to the elements in the `set`. Just as we cannot change the key part of a `map` element, the keys in a `set` are also `const`. We can use a `set` iterator to read, but not write, an element's value:

Click here to view code image

```
set<int> iset = {0,1,2,3,4,5,6,7,8,9};
set<int>::iterator set_it = iset.begin();
if (set_it != iset.end()) {
    *set_it = 42;              // error: keys in a set are read-only
    cout << *set_it << endl; // ok: can read the key
}
```

Iterating across an Associative Container

The `map` and `set` types provide all the `begin` and `end` operations from Table 9.2 (p. 330). As usual, we can use these functions to obtain iterators that we can use to traverse the container. For example, we can rewrite the loop that printed the results in our word-counting program on page 421 as follows:

Click here to view code image

```
// get an iterator positioned on the first element
auto map_it = word_count.cbegin();
// compare the current iterator to the off-the-end iterator
while (map_it != word_count.cend()) {
    // dereference the iterator to print the element key--value pairs
    cout << map_it->first << " occurs "
         << map_it->second << " times" << endl;
    ++map_it;  // increment the iterator to denote the next element
}
```

The `while` condition and increment for the iterator in this loop look a lot like the programs we wrote that printed the contents of a `vector` or a `string`. We initialize an iterator, `map_it`, to refer to the first element in `word_count`. As long as the iterator is not equal to the `end` value, we print the current element and then increment the iterator. The output statement dereferences `map_it` to get the members of `pair` but is otherwise the same as the one in our original program.

 **Note**

The output of this program is in alphabetical order. When we use an iterator to traverse a `map`, `multimap`, `set`, or `multiset`, the iterators yield elements in ascending key order.

Associative Containers and Algorithms

In general, we do not use the generic algorithms (Chapter 10) with the associative containers. The fact that the keys are `const` means that we cannot pass associative container iterators to algorithms that write to or reorder container elements. Such algorithms need to write to the elements. The elements in the `set` types are `const`, and those in `map`s are `pair`s whose first element is `const`.

Associative containers can be used with the algorithms that read elements. However, many of these algorithms search the sequence. Because elements in an associative container can be found (quickly) by their key, it is almost always a bad idea to use a generic search algorithm. For example, as we'll see in § 11.3.5 (p. 436), the associative containers define a member named `find`, which directly fetches the element with a given key. We could use the generic `find` algorithm to look for an element, but that algorithm does a sequential search. It is much faster to use the `find` member defined by the container than to call the generic version.

In practice, if we do so at all, we use an associative container with the algorithms either as the source sequence or as a destination. For example, we might use the generic `copy` algorithm to copy the elements from an associative container into another sequence. Similarly, we can call `inserter` to bind an insert iterator (§ 10.4.1, p. 401) to an associative container. Using `inserter`, we can use the associative container as a destination for another algorithm.

Exercises Section 11.3.1

Exercise 11.15: What are the `mapped_type`, `key_type`, and `value_type` of a `map` from `int` to `vector<int>`?

Exercise 11.16: Using a `map` iterator write an expression that assigns a value to an element.

Exercise 11.17: Assuming `c` is a `multiset` of `strings` and `v` is a `vector` of `strings`, explain the following calls. Indicate whether each call is legal:

Click here to view code image

```
copy(v.begin(), v.end(), inserter(c, c.end()));
copy(v.begin(), v.end(), back_inserter(c));
copy(c.begin(), c.end(), inserter(v, v.end()));
copy(c.begin(), c.end(), back_inserter(v));
```

Exercise 11.18: Write the type of `map_it` from the loop on page 430 without using `auto` or `decltype`.

Exercise 11.19: Define a variable that you initialize by calling `begin()` on the `multiset` named `bookstore` from § 11.2.2 (p. 425). Write the variable's type without using `auto` or `decltype`.

11.3.2. Adding Elements

The `insert` members (Table 11.4 (overleaf)) add one element or a range of elements. Because `map` and `set` (and the corresponding unordered types) contain unique keys, inserting an element that is already present has no effect:

Click here to view code image

```
vector<int> ivec = {2,4,6,8,2,4,6,8};      // ivec has eight elements
set<int> set2;                              // empty set
set2.insert(ivec.cbegin(), ivec.cend()); // set2 has four elements
set2.insert({1,3,5,7,1,3,5,7});        // set2 now has eight elements
```

Table 11.4. Associative Container insert Operations

`c.insert(v)`	`v` `value_type` object; *args* are used to construct an element.
`c.emplace(`*args*`)`	For `map` and `set`, the element is inserted (or constructed) only if an element with the given key is not already in `c`. Returns a `pair` containing an iterator referring to the element with the given key and a `bool` indicating whether the element was inserted. For `multimap` and `multiset`, inserts (or constructs) the given element and returns an iterator to the new element.
`c.insert(b, e)`	`b` and `e` are iterators that denote a range of `c::value_type` values;
`c.insert(il)`	`il` is a braced list of such values. Returns `void`. For `map` and `set`, inserts the elements with keys that are not already in `c`. For `multimap` and `multiset` inserts, each element in the range.
`c.insert(p, v)`	Like `insert(v)` (or `emplace(`*args*`)`), but uses iterator `p` as a hint
`c.emplace(p,` *args*`)`	for where to begin the search for where the new element should be stored. Returns an iterator to the element with the given key.

The versions of `insert` that take a pair of iterators or an initializer list work similarly to the corresponding constructors (§ 11.2.1, p. 423)—only the first element with a given key is inserted.

Adding Elements to a map

When we `insert` into a `map`, we must remember that the element type is a `pair`. Often, we don't have a `pair` object that we want to insert. Instead, we create a

pair in the argument list to insert:

Click here to view code image

```
// four ways to add word to word_count
word_count.insert({word, 1});
word_count.insert(make_pair(word, 1));
word_count.insert(pair<string, size_t>(word, 1));
word_count.insert(map<string, size_t>::value_type(word, 1));
```

C++ 11

As we've seen, under the new standard the easiest way to create a pair is to use brace initialization inside the argument list. Alternatively, we can call make_pair or explicitly construct the pair. The argument in the last call to insert:

Click here to view code image

```
map<string, size_t>::value_type(s, 1)
```

constructs a new object of the appropriate pair type to insert into the map.

Testing the Return from insert

The value returned by insert (or emplace) depends on the container type and the parameters. For the containers that have unique keys, the versions of insert and emplace that add a single element return a pair that lets us know whether the insertion happened. The first member of the pair is an iterator to the element with the given key; the second is a bool indicating whether that element was inserted, or was already there. If the key is already in the container, then insert does nothing, and the bool portion of the return value is false. If the key isn't present, then the element is inserted and the bool is true.

As an example, we'll rewrite our word-counting program to use insert:

Click here to view code image

```
// more verbose way to count number of times each word occurs in the input
map<string, size_t> word_count; // empty map from string to size_t
string word;
while (cin >> word) {
    // inserts an element with key equal to word and value 1;
    // if word is already in word_count, insert does nothing
    auto ret = word_count.insert({word, 1});
    if (!ret.second)          // word was already in word_count
        ++ret.first->second;  // increment the counter
}
```

For each word, we attempt to insert it with a value 1. If word is already in the map, then nothing happens. In particular, the counter associated with word is unchanged. If word is not already in the map, then that string is added to the map

and its counter value is set to 1.

The `if` test examines the `bool` part of the return value. If that value is `false`, then the insertion didn't happen. In this case, `word` was already in `word_count`, so we must increment the value associated with that element.

Unwinding the Syntax

The statement that increments the counter in this version of the word-counting program can be hard to understand. It will be easier to understand that expression by first parenthesizing it to reflect the precedence (§ 4.1.2, p. 136) of the operators:

Click here to view code image

```
++((ret.first)->second); // equivalent expression
```

Explaining this expression step by step:

ret holds the value returned by `insert`, which is a `pair`.

ret.first is the `first` member of that `pair`, which is a `map` iterator referring to the element with the given key.

ret.first-> dereferences that iterator to fetch that element. Elements in the `map` are also `pairs`.

ret.first->second is the value part of the map element `pair`.

++ret.first->second increments that value.

Putting it back together, the increment statement fetches the iterator for the element with the key `word` and increments the counter associated with the key we tried to insert.

For readers using an older compiler or reading code that predates the new standard, declaring and initializing `ret` is also somewhat tricky:

Click here to view code image

```
pair<map<string, size_t>::iterator, bool> ret =
             word_count.insert(make_pair(word, 1));
```

It should be easy to see that we're defining a `pair` and that the second type of the `pair` is `bool`. The first type of that `pair` is a bit harder to understand. It is the `iterator` type defined by the `map<string, size_t>` type.

Adding Elements to multiset or multimap

Our word-counting program depends on the fact that a given key can occur only once. That way, there is only one counter associated with any given word. Sometimes, we want to be able to add additional elements with the same key. For example, we might want to map authors to titles of the books they have written. In

this case, there might be multiple entries for each author, so we'd use a `multimap` rather than a `map`. Because keys in a `multi` container need not be unique, `insert` on these types always inserts an element:

Click here to view code image

```
multimap<string, string> authors;
// adds the first element with the key Barth, John
authors.insert({"Barth, John", "Sot-Weed Factor"});
// ok: adds the second element with the key Barth, John
authors.insert({"Barth, John", "Lost in the Funhouse"});
```

For the containers that allow multiple keys, the `insert` operation that takes a single element returns an iterator to the new element. There is no need to return a `bool`, because `insert` always adds a new element in these types.

Exercises Section 11.3.2

Exercise 11.20: Rewrite the word-counting program from § 11.1 (p. 421) to use `insert` instead of subscripting. Which program do you think is easier to write and read? Explain your reasoning.

Exercise 11.21: Assuming `word_count` is a `map` from `string` to `size_t` and `word` is a `string`, explain the following loop:

Click here to view code image

```
while (cin >> word)
  ++word_count.insert({word, 0}).first->second;
```

Exercise 11.22: Given a `map<string, vector<int>>`, write the types used as an argument and as the return value for the version of `insert` that inserts one element.

Exercise 11.23: Rewrite the `map` that stored `vectors` of children's names with a key that is the family last name for the exercises in § 11.2.1 (p. 424) to use a `multimap`.

11.3.3. Erasing Elements

The associative containers define three versions of `erase`, which are described in Table 11.5. As with the sequential containers, we can `erase` one element or a range of elements by passing `erase` an iterator or an iterator pair. These versions of `erase` are similar to the corresponding operations on sequential containers: The indicated element(s) are removed and the function returns `void`.

Table 11.5. Removing Elements from an Associative Container

`c.erase(k)`	Removes every element with key `k` from `c`. Returns `size_type` indicating the number of elements removed.
`c.erase(p)`	Removes the element denoted by the iterator `p` from `c`. `p` must refer to an actual element in `c`; it must not be equal to `c.end()`. Returns an iterator to the element after `p` or `c.end()` if `p` denotes the last element in `c`.
`c.erase(b, e)`	Removes the elements in the range denoted by the iterator pair `b`, `e`. Returns `e`.

The associative containers supply an additional `erase` operation that takes a `key_type` argument. This version removes all the elements, if any, with the given key and returns a count of how many elements were removed. We can use this version to remove a specific word from `word_count` before printing the results:

Click here to view code image

```
// erase on a key returns the number of elements removed
if (word_count.erase(removal_word))
     cout << "ok: " << removal_word << " removed\n";
else cout << "oops: " << removal_word << " not found!\n";
```

For the containers with unique keys, the return from `erase` is always either zero or one. If the return value is zero, then the element we wanted to erase was not in the container.

For types that allow multiple keys, the number of elements removed could be greater than one:

Click here to view code image

```
auto cnt = authors.erase("Barth, John");
```

If `authors` is the `multimap` we created in § 11.3.2 (p. 434), then `cnt` will be 2.

11.3.4. Subscripting a `map`

The `map` and `unordered_map` containers provide the subscript operator and a corresponding `at` function (§ 9.3.2, p. 348), which are described in Table 11.6 (overleaf). The `set` types do not support subscripting because there is no "value" associated with a key in a `set`. The elements are themselves keys, so the operation of "fetching the value associated with a key" is meaningless. We cannot subscript a `multimap` or an `unordered_multimap` because there may be more than one value associated with a given key.

Table 11.6. Subscript Operation for `map` and `unordered_map`

`c[k]`	Returns the element with key `k`; if `k` is not in `c`, adds a new, value-initialized element with key `k`.
`c.at(k)`	Checked access to the element with key `k`; throws an `out_of_range` exception (§ 5.6, p. 193) if `k` is not in `c`.

Like the other subscript operators we've used, the `map` subscript takes an index (that is, a key) and fetches the value associated with that key. However, unlike other subscript operators, if the key is not already present, *a new element is created and inserted* into the `map` for that key. The associated value is value initialized (§ 3.3.1, p. 98).

For example, when we write

Click here to view code image

```
map <string, size_t> word_count; // empty map
// insert a value-initialized element with key Anna; then assign 1 to its value
word_count["Anna"] = 1;
```

the following steps take place:

- `word_count` is searched for the element whose key is `Anna`. The element is not found.
- A new key-value pair is inserted into `word_count`. The key is a `const string` holding `Anna`. The value is value initialized, meaning in this case that the value is 0.
- The newly inserted element is fetched and is given the value 1.

Because the subscript operator might insert an element, we may use subscript only on a `map` that is not `const`.

Note

Subscripting a `map` behaves quite differently from subscripting an array or `vector`: Using a key that is not already present *adds* an element with that key to the `map`.

Using the Value Returned from a Subscript Operation

Another way in which the `map` subscript differs from other subscript operators we've used is its return type. Ordinarily, the type returned by dereferencing an iterator and the type returned by the subscript operator are the same. Not so for `maps`: when we subscript a `map`, we get a `mapped_type` object; when we dereference a `map` iterator, we get a `value_type` object (§ 11.3, p. 428).

In common with other subscripts, the `map` subscript operator returns an lvalue (§ 4.1.1, p. 135). Because the return is an lvalue, we can read or write the element:

Click here to view code image

```
cout << word_count["Anna"]; // fetch the element indexed by Anna; prints 1
++word_count["Anna"];       // fetch the element and add 1 to it
cout << word_count["Anna"]; // fetch the element and print it; prints 2
```

Note

Unlike `vector` or `string`, the type returned by the `map` subscript operator differs from the type obtained by dereferencing a `map` iterator.

The fact that the subscript operator adds an element if it is not already in the `map` allows us to write surprisingly succinct programs such as the loop inside our word-counting program (§ 11.1, p. 421). On the other hand, sometimes we only want to know whether an element is present and *do not* want to add the element if it is not. In such cases, we must not use the subscript operator.

11.3.5. Accessing Elements

The associative containers provide various ways to find a given element, which are described in Table 11.7 (p. 438). Which operation to use depends on what problem we are trying to solve. If all we care about is whether a particular element is in the container, it is probably best to use `find`. For the containers that can hold only unique keys, it probably doesn't matter whether we use `find` or `count`. However, for the containers with multiple keys, `count` has to do more work: If the element is present, it still has to count how many elements have the same key. If we don't need the count, it's best to use `find`:

Table 11.7. Operations to Find Elements in an Associative Container

lower_bound and upper_bound not valid for the unordered containers. Subscript and at operations only for map and unordered_map that are not const.

`c.find(k)`	Returns an iterator to the (first) element with key `k`, or the off-the-end iterator if `k` is not in the container.
`c.count(k)`	Returns the number of elements with key `k`. For the containers with unique keys, the result is always zero or one.
`c.lower_bound(k)`	Returns an iterator to the first element with key not less than `k`.
`c.upper_bound(k)`	Returns an iterator to the first element with key greater than `k`.
`c.equal_range(k)`	Returns a `pair` of iterators denoting the elements with key `k`. If `k` is not present, both members are `c.end()`.

Exercises Section 11.3.4

Exercise 11.24: What does the following program do?

```
map<int, int> m;
m[0] = 1;
```

Exercise 11.25: Contrast the following program with the one in the previous exercise

```
vector<int> v;
v[0] = 1;
```

Exercise 11.26: What type can be used to subscript a `map`? What type does the subscript operator return? Give a concrete example—that is, define a `map` and then write the types that can be used to subscript the `map` and the type that would be returned from the subscript operator.

Click here to view code image

```
set<int> iset = {0,1,2,3,4,5,6,7,8,9};
iset.find(1);    // returns an iterator that refers to the element with key == 1
iset.find(11);   // returns the iterator == iset.end()
iset.count(1);   // returns 1
iset.count(11);  // returns 0
```

Using find Instead of Subscript for maps

For the `map` and `unordered_map` types, the subscript operator provides the simplest method of retrieving a value. However, as we've just seen, using a subscript has an important side effect: If that key is not already in the `map`, then subscript inserts an element with that key. Whether this behavior is correct depends on our expectations. Our word-counting programs relied on the fact that using a nonexistent key as a

subscript inserts an element with that key and value 0.

Sometimes, we want to know if an element with a given key is present without changing the `map`. We cannot use the subscript operator to determine whether an element is present, because the subscript operator inserts a new element if the key is not already there. In such cases, we should use `find`:

Click here to view code image

```
if (word_count.find("foobar") == word_count.end())
    cout << "foobar is not in the map" << endl;
```

Finding Elements in a multimap or multiset

Finding an element in an associative container that requires unique keys is a simple matter—the element is or is not in the container. For the containers that allow multiple keys, the process is more complicated: There may be many elements with the given key. When a `multimap` or `multiset` has multiple elements of a given key, those elements will be adjacent within the container.

For example, given our map from author to titles, we might want to print all the books by a particular author. We can solve this problem in three different ways. The most obvious way uses `find` and `count`:

Click here to view code image

```
string search_item("Alain de Botton"); // author we'll look for
auto entries = authors.count(search_item); // number of elements
auto iter = authors.find(search_item); // first entry for this author
// loop through the number of entries there are for this author
while(entries) {
    cout << iter->second << endl; // print each title
    ++iter;     // advance to the next title
    --entries;  // keep track of how many we've printed
}
```

We start by determining how many entries there are for the author by calling `count` and getting an iterator to the first element with this key by calling `find`. The number of iterations of the `for` loop depends on the number returned from `count`. In particular, if the `count` was zero, then the loop is never executed.

Note

We are guaranteed that iterating across a `multimap` or `multiset` returns all the elements with a given key in sequence.

A Different, Iterator-Oriented Solution

Alternatively, we can solve our problem using `lower_bound` and `upper_bound`. Each of these operations take a key and returns an iterator. If the key is in the container, the iterator returned from `lower_bound` will refer to the first instance of that key and the iterator returned by `upper_bound` will refer just after the last instance of the key. If the element is not in the `multimap`, then `lower_bound` and `upper_bound` will return equal iterators; both will refer to the point at which the key can be inserted without disrupting the order. Thus, calling `lower_bound` and `upper_bound` on the same key yields an iterator range (§ 9.2.1, p. 331) that denotes all the elements with that key.

Of course, the iterator returned from these operations might be the off-the-end iterator for the container itself. If the element we're looking for has the largest key in the container, then `upper_bound` on that key returns the off-the-end iterator. If the key is not present and is larger than any key in the container, then the return from `lower_bound` will also be the off-the-end iterator.

Note

The iterator returned from `lower_bound` may or may not refer to an element with the given key. If the key is not in the container, then `lower_bound` refers to the first point at which this key can be inserted while preserving the element order within the container.

Using these operations, we can rewrite our program as follows:

Click here to view code image

```
// definitions of authors and search_item as above
// beg and end denote the range of elements for this author
for (auto beg = authors.lower_bound(search_item),
          end = authors.upper_bound(search_item);
     beg != end; ++beg)
    cout << beg->second << endl; // print each title
```

This program does the same work as the previous one that used `count` and `find` but accomplishes its task more directly. The call to `lower_bound` positions `beg` so that it refers to the first element matching `search_item` if there is one. If there is no such element, then `beg` refers to the first element with a key larger than `search_item`, which could be the off-the-end iterator. The call to `upper_bound` sets `end` to refer to the element just beyond the last element with the given key. These operations say nothing about whether the key is present. The important point is that the return values act like an iterator range (§ 9.2.1, p. 331).

If there is no element for this key, then `lower_bound` and `upper_bound` will be

equal. Both will refer to the point at which this key can be inserted while maintaining the container order.

Assuming there are elements with this key, `beg` will refer to the first such element. We can increment `beg` to traverse the elements with this key. The iterator in `end` will signal when we've seen all the elements. When `beg` equals `end`, we have seen every element with this key.

Because these iterators form a range, we can use a `for` loop to traverse that range. The loop is executed zero or more times and prints the entries, if any, for the given author. If there are no elements, then `beg` and `end` are equal and the loop is never executed. Otherwise, we know that the increment to `beg` will eventually reach `end` and that in the process we will print each record associated with this author.

Note

If `lower_bound` and `upper_bound` return the same iterator, then the given key is not in the container.

The equal_range Function

The remaining way to solve this problem is the most direct of the three approaches: Instead of calling `upper_bound` and `lower_bound`, we can call `equal_range`.

This function takes a key and returns a `pair` of iterators. If the key is present, then the first iterator refers to the first instance of the key and the second iterator refers one past the last instance of the key. If no matching element is found, then both the first and second iterators refer to the position where this key can be inserted.

We can use `equal_range` to modify our program once again:

Click here to view code image

```
// definitions of authors and search_item as above
// pos holds iterators that denote the range of elements for this key
for (auto pos = authors.equal_range(search_item);
     pos.first != pos.second; ++pos.first)
    cout << pos.first->second << endl; // print each title
```

This program is essentially identical to the previous one that used `upper_bound` and `lower_bound`. Instead of using local variables, `beg` and `end`, to hold the iterator range, we use the `pair` returned by `equal_range`. The `first` member of that `pair` holds the same iterator as `lower_bound` would have returned and `second` holds the iterator `upper_bound` would have returned. Thus, in this program `pos.first` is equivalent to `beg`, and `pos.second` is equivalent to `end`.

Exercises Section 11.3.5

Exercise 11.27: What kinds of problems would you use `count` to solve? When might you use `find` instead?

Exercise 11.28: Define and initialize a variable to hold the result of calling `find` on a `map` from `string` to `vector` of `int`.

Exercise 11.29: What do `upper_bound`, `lower_bound`, and `equal_range` return when you pass them a key that is not in the container?

Exercise 11.30: Explain the meaning of the operand `pos.first->second` used in the output expression of the final program in this section.

Exercise 11.31: Write a program that defines a `multimap` of authors and their works. Use `find` to find an element in the `multimap` and `erase` that element. Be sure your program works correctly if the element you look for is not in the `map`.

Exercise 11.32: Using the `multimap` from the previous exercise, write a program to print the list of authors and their works alphabetically.

11.3.6. A Word Transformation Map

We'll close this section with a program to illustrate creating, searching, and iterating across a `map`. We'll write a program that, given one `string`, transforms it into another. The input to our program is two files. The first file contains rules that we will use to transform the text in the second file. Each rule consists of a word that might be in the input file and a phrase to use in its place. The idea is that whenever the first word appears in the input, we will replace it with the corresponding phrase. The second file contains the text to transform.

If the contents of the word-transformation file are

```
brb be right back
k okay?
y why
r are
u you
pic picture
thk thanks!
l8r later
```

and the text we are given to transform is

```
where r u
y dont u send me a pic
k thk l8r
```

then the program should generate the following output:

where are you
why dont you send me a picture
okay? thanks! later

The Word Transformation Program

Our solution will use three functions. The `word_transform` function will manage the overall processing. It will take two `ifstream` arguments: The first will be bound to the word-transformation file and the second to the file of text we're to transform. The `buildMap` function will read the file of transformation rules and create a `map` from each word to its transformation. The `transform` function will take a `string` and return the transformation if there is one.

We'll start by defining the `word_transform` function. The important parts are the calls to `buildMap` and `transform`:

Click here to view code image

```
void word_transform(ifstream &map_file, ifstream &input)
{
    auto trans_map = buildMap(map_file); // store the transformations
    string text;                         // hold each line from the input
    while (getline(input, text)) {   // read a line of input
        istringstream stream(text); // read each word
        string word;
         bool firstword = true;       // controls whether a space is
printed
        while (stream >> word) {
           if (firstword)
               firstword = false;
           else
               cout << " ";   // print a space between words
           // transform returns its first argument or its transformation
           cout << transform(word, trans_map); // print the output
        }
        cout << endl;           // done with this line of input
    }
}
```

The function starts by calling `buildMap` to generate the word-transformation `map`. We store the result in `trans_map`. The rest of the function processes the `input` file. The `while` loop uses `getline` to read the input file a line at a time. We read by line so that our output will have line breaks at the same position as in the input file. To get the words from each line, we use a nested `while` loop that uses an `istringstream` (§ 8.3, p. 321) to process each word in the current line.

The inner `while` prints the output using the `bool firstword` to determine whether to print a space. The call to `transform` obtains the word to print. The value

returned from `transform` is either the original `string` in `word` or its corresponding transformation from `trans_map`.

Building the Transformation Map

The `buildMap` function reads its given file and builds the transformation `map`.

Click here to view code image

```
map<string, string> buildMap(ifstream &map_file)
{
    map<string, string> trans_map;   // holds the transformations
    string key;     // a word to transform
    string value;   // phrase to use instead
    // read the first word into key and the rest of the line into value
    while (map_file >> key && getline(map_file, value))
        if (value.size() > 1) // check that there is a transformation
            trans_map[key] = value.substr(1); // skip leading
space
        else
            throw runtime_error("no rule for " + key);
    return trans_map;
}
```

Each line in `map_file` corresponds to a rule. Each rule is a word followed by a phrase, which might contain multiple words. We use >> to read the word that we will transform into `key` and call `getline` to read the rest of the line into `value`. Because `getline` does not skip leading spaces (§ 3.2.2, p. 87), we need to skip the space between the word and its corresponding rule. Before we store the transformation, we check that we got more than one character. If so, we call `substr` (§ 9.5.1, p. 361) to skip the space that separated the transformation phrase from its corresponding word and store that substring in `trans_map`,

Note that we use the subscript operator to add the key–value pairs. Implicitly, we are ignoring what should happen if a word appears more than once in our transformation file. If a word does appear multiple times, our loops will put the last corresponding phrase into `trans_map`. When the `while` concludes, `trans_map` contains the data that we need to transform the input.

Generating a Transformation

The `transform` function does the actual transformation. Its parameters are references to the `string` to transform and to the transformation `map`. If the given `string` is in the `map`, `transform` returns the corresponding transformation. If the given `string` is not in the `map`, `transform` returns its argument:

Click here to view code image

```
const string &
transform(const string &s, const map<string, string> &m)
{
    // the actual map work; this part is the heart of the program
    auto map_it = m.find(s);
    // if this word is in the transformation map
    if (map_it != m.cend())
        return map_it->second; // use the replacement word
    else
        return s;              // otherwise return the original unchanged
}
```

We start by calling `find` to determine whether the given `string` is in the `map`. If it is, then `find` returns an iterator to the corresponding element. Otherwise, `find` returns the off-the-end iterator. If the element is found, we dereference the iterator, obtaining a `pair` that holds the key and value for that element (§ 11.3, p. 428). We return the `second` member, which is the transformation to use in place of `s`.

Exercises Section 11.3.6

Exercise 11.33: Implement your own version of the word-transformation program.

Exercise 11.34: What would happen if we used the subscript operator instead of `find` in the `transform` function?

Exercise 11.35: In `buildMap`, what effect, if any, would there be from rewriting

Click here to view code image

```
    trans_map[key] = value.substr(1);
as trans_map.insert({key, value.substr(1)})?
```

Exercise 11.36: Our program does no checking on the validity of either input file. In particular, it assumes that the rules in the transformation file are all sensible. What would happen if a line in that file has a key, one space, and then the end of the line? Predict the behavior and then check it against your version of the program.

11.4. The Unordered Containers

The new standard defines four **unordered associative containers**. Rather than using a comparison operation to organize their elements, these containers use a *hash*

function and the key type's == operator. An unordered container is most useful when we have a key type for which there is no obvious ordering relationship among the elements. These containers are also useful for applications in which the cost of maintaining the elements in order is prohibitive.

Although hashing gives better average case performance in principle, achieving good results in practice often requires a fair bit of performance testing and tweaking. As a result, it is usually easier (and often yields better performance) to use an ordered container.

Tip

Use an unordered container if the key type is inherently unordered or if performance testing reveals problems that hashing might solve.

Using an Unordered Container

Aside from operations that manage the hashing, the unordered containers provide the same operations (`find`, `insert`, and so on) as the ordered containers. That means that the operations we've used on `map` and `set` apply to `unordered_map` and `unordered_set` as well. Similarly for the unordered versions of the containers that allow multiple keys.

As a result, we can usually use an unordered container in place of the corresponding ordered container, and vice versa. However, because the elements are not stored in order, the output of a program that uses an unordered container will (ordinarily) differ from the same program using an ordered container.

For example, we can rewrite our original word-counting program from § 11.1 (p. 421) to use an `unordered_map`:

Click here to view code image

```
// count occurrences, but the words won't be in alphabetical order
unordered_map<string, size_t> word_count;
string word;
while (cin >> word)
    ++word_count[word]; // fetch and increment the counter for word
for (const auto &w : word_count) // for each element in the map
     // print the results
     cout <<  w.first << " occurs " << w.second
          << ((w.second > 1) ? " times" : " time") << endl;
```

The type of `word_count` is the only difference between this program and our original. If we run this version on the same input as our original program,

containers. occurs 1 time
use occurs 1 time
can occurs 1 time
examples occurs 1 time
...

we'll obtain the same count for each word in the input. However, the output is unlikely to be in alphabetical order.

Managing the Buckets

The unordered containers are organized as a collection of buckets, each of which holds zero or more elements. These containers use a hash function to map elements to buckets. To access an element, the container first computes the element's hash code, which tells which bucket to search. The container puts all of its elements with a given hash value into the same bucket. If the container allows multiple elements with a given key, all the elements with the same key will be in the same bucket. As a result, the performance of an unordered container depends on the quality of its hash function and on the number and size of its buckets.

The hash function must always yield the same result when called with the same argument. Ideally, the hash function also maps each particular value to a unique bucket. However, a hash function is allowed to map elements with differing keys to the same bucket. When a bucket holds several elements, those elements are searched sequentially to find the one we want. Typically, computing an element's hash code and finding its bucket is a fast operation. However, if the bucket has many elements, many comparisons may be needed to find a particular element.

The unordered containers provide a set of functions, listed in Table 11.8, that let us manage the buckets. These members let us inquire about the state of the container and force the container to reorganize itself as needed.

Table 11.8. Unordered Container Management Operations

Bucket Interface	
`c.bucket_count()`	Number of buckets in use.
`c.max_bucket_count()`	Largest number of buckets this container can hold.
`c.bucket_size(n)`	Number of elements in the nth bucket.
`c.bucket(k)`	Bucket in which elements with key `k` would be found.
Bucket Iteration	
`local_iterator`	Iterator type that can access elements in a bucket.
`const_local_iterator`	`const` version of the bucket iterator.
`c.begin(n), c.end(n)`	Iterator to the first, one past the last element in bucket `n`.
`c.cbegin(n), c.cend(n)`	Returns `const_local_iterator`.
Hash Policy	
`c.load_factor()`	Average number of elements per bucket. Returns `float`.
`c.max_load_factor()`	Average bucket size that `c` tries to maintain. `c` adds buckets to keep `load_factor <= max_load_factor`. Returns `float`.
`c.rehash(n)`	Reorganize storage so that `bucket_count >= n` and and `bucket_count > size/max_load_factor`.
`c.reserve(n)`	Reorganize so that `c` can hold `n` elements without a `rehash`.

Requirements on Key Type for Unordered Containers

By default, the unordered containers use the == operator on the key type to compare elements. They also use an object of type `hash<key_type>` to generate the hash code for each element. The library supplies versions of the **hash** template for the built-in types, including pointers. It also defines `hash` for some of the library types, including `strings` and the smart pointer types that we will describe in Chapter 12. Thus, we can directly define unordered containers whose key is one of the built-in types (including pointer types), or a `string`, or a smart pointer.

However, we cannot directly define an unordered container that uses a our own class types for its key type. Unlike the containers, we cannot use the hash template directly. Instead, we must supply our own version of the `hash` template. We'll see how to do so in § 16.5 (p. 709).

Instead of using the default `hash`, we can use a strategy similar to the one we used to override the default comparison operation on keys for the ordered containers (§ 11.2.2, p. 425). To use `Sales_data` as the key, we'll need to supply functions to replace both the == operator and to calculate a hash code. We'll start by defining these functions:

Click here to view code image

```
size_t hasher(const Sales_data &sd)
{
    return hash<string>()(sd.isbn());
}
bool eqOp(const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.isbn() == rhs.isbn();
```

```
}
```

Our `hasher` function uses an object of the library `hash` of `string` type to generate a hash code from the ISBN member. Similarly, the `eqOp` funciton compares two `Sales_data` objects by comparing their ISBNs.

We can use these functions to define an `unordered_multiset` as follows

Click here to view code image

```
using SD_multiset = unordered_multiset<Sales_data,
                                       decltype(hasher)*, decltype(eqOp)*>;
// arguments are the bucket size and pointers to the hash function and equality operator
SD_multiset bookstore(42, hasher, eqOp);
```

To simplify the declaration of `bookstore` we first define a type alias (§ 2.5.1, p. 67) for an `unordered_multiset` whose hash and equality operations have the same types as our `hasher` and `eqOp` functions. Using that type, we define `bookstore` passing pointers to the functions we want `bookstore` to use.

If our class has its own == operator we can override just the hash function:

Click here to view code image

```
// use FooHash to generate the hash code; Foo must have an == operator
unordered_set<Foo, decltype(FooHash)*> fooSet(10, FooHash);
```

Exercises Section 11.4

Exercise 11.37: What are the advantages of an unordered container as compared to the ordered version of that container? What are the advantages of the ordered version?

Exercise 11.38: Rewrite the word-counting (§ 11.1, p. 421) and word-transformation (§ 11.3.6, p. 440) programs to use an `unordered_map`.

Chapter Summary

The associative containers support efficient lookup and retrieval of elements by key. The use of a key distinguishes the associative containers from the sequential containers, in which elements are accessed positionally.

There are eight associative containers, each of which

- Is a `map` or a `set`. a `map` stores key-value pairs; a `set` stores only keys.
- Requires unique keys or not.
- Keeps keys in order or not.

Ordered containers use a comparison function to order the elements by key. By

default, the comparison is the < operator on the keys. Unordered containers use the key type's == operator and an object of type `hash<key_type>` to organize their elements.

Containers with nonunique keys include the word `multi` in their names; those that use hashing start with the word `unordered`. A `set` is an ordered collection in which each key may appear only once; an `unordered_multiset` is an unordered collection of keys in which the keys can appear multiple times.

The associative containers share many operations with the sequential containers. However, the associative containers define some new operations and redefine the meaning or return types of some operations common to both the sequential and associative containers. The differences in the operations reflect the use of keys in associative containers.

Iterators for the ordered containers access elements in order by key. Elements with the same key are stored adjacent to one another in both the ordered and unordered containers.

Defined Terms

associative array Array whose elements are indexed by key rather than positionally. We say that the array maps a key to its associated value.

associative container Type that holds a collection of objects that supports efficient lookup by key.

hash Special library template that the unordered containers use to manage the position of their elements.

hash function Function that maps values of a given type to integral (`size_t`) values. Equal values must map to equal integers; unequal values should map to unequal integers where possible.

key_type Type defined by the associative containers that is the type for the keys used to store and retrieve values. For a `map`, `key_type` is the type used to index the `map`. For `set`, `key_type` and `value_type` are the same.

map Associative container type that defines an associative array. Like `vector`, `map` is a class template. A `map`, however, is defined with two types: the type of the key and the type of the associated value. In a `map`, a given key may appear only once. Each key is associated with a particular value. Dereferencing a `map` iterator yields a `pair` that holds a `const` key and its associated value.

mapped_type Type defined by map types that is the type of the values associated with the keys in the map.

multimap Associative container similar to `map` except that in a `multimap`, a

given key may appear more than once. `multimap` does not support subscripting.

multiset Associative container type that holds keys. In a `multiset`, a given key may appear more than once.

pair Type that holds two `public` data members named `first` and `second`. The `pair` type is a template type that takes two type parameters that are used as the types of these members.

set Associative container that holds keys. In a `set`, a given key may appear only once.

strict weak ordering Relationship among the keys used in an associative container. In a strict weak ordering, it is possible to compare any two values and determine which of the two is less than the other. If neither value is less than the other, then the two values are considered equal.

unordered container Associative containers that use hashing rather than a comparison operation on keys to store and access elements. The performance of these containers depends on the quality of the hash function.

unordered_map Container with elements that are key–value pairs, permits only one element per key.

unordered_multimap Container with elements that are key–value pairs, allows multiple elements per key.

unordered_multiset Container that stores keys, allows multiple elements per key.

unordered_set Container that stores keys, permits only one element per key.

value_type Type of the element stored in a container. For `set` and `multiset`, `value_type` and `key_type` are the same. For `map` and `multimap`, this type is a `pair` whose `first` member has type `const key_type` and whose `second` member has type `mapped_type`.

*** operator** Dereference operator. When applied to a `map`, `set`, `multimap`, or `multiset` iterator `*` yields a `value_type`. Note, that for `map` and `multimap`, the `value_type` is a `pair`.

[ ] operator Subscript operator. Defined only for nonconst obejcts of type `map` and `unordered_map`. For the map types, `[]` takes an index that must be a `key_type` (or type that can be converted to `key_type`). Yields a `mapped_type` value.

Chapter 12. Dynamic Memory

Contents

The programs we've written so far have used objects that have well-defined lifetimes. Global objects are allocated at program start-up and destroyed when the program ends. Local, automatic objects are created and destroyed when the block in which they are defined is entered and exited. Local `static` objects are allocated before their first use and are destroyed when the program ends.

In addition to supporting automatic and `static` objects, C++ lets us allocate objects dynamically. Dynamically allocated objects have a lifetime that is independent of where they are created; they exist until they are explicitly freed.

Properly freeing dynamic objects turns out to be a surprisingly rich source of bugs. To make using dynamic objects safer, the library defines two smart pointer types that manage dynamically allocated objects. Smart pointers ensure that the objects to which they point are automatically freed when it is appropriate to do so.

Our programs have used only static or stack memory. Static memory is used for local `static` objects (§ 6.1.1, p. 205), for class `static` data members (§ 7.6, p. 300), and for variables defined outside any function. Stack memory is used for nonstatic objects defined inside functions. Objects allocated in static or stack memory are automatically created and destroyed by the compiler. Stack objects exist only while the block in which they are defined is executing; `static` objects are allocated before they are used, and they are destroyed when the program ends.

In addition to static or stack memory, every program also has a pool of memory that it can use. This memory is referred to as the **free store** or **heap**. Programs use the heap for objects that they **dynamically allocate**—that is, for objects that the program allocates at run time. The program controls the lifetime of dynamic objects; our code must explicitly destroy such objects when they are no longer needed.

Warning

Although necessary at times, dynamic memory is notoriously tricky to manage correctly.

12.1. Dynamic Memory and Smart Pointers

In C++, dynamic memory is managed through a pair of operators: **new**, which allocates, and optionally initializes, an object in dynamic memory and returns a pointer to that object; and **delete**, which takes a pointer to a dynamic object, destroys that object, and frees the associated memory.

Dynamic memory is problematic because it is surprisingly hard to ensure that we free memory at the right time. Either we forget to free the memory—in which case we have a memory leak—or we free the memory when there are still pointers referring to that memory—in which case we have a pointer that refers to memory that is no longer valid.

To make using dynamic memory easier (and safer), the new library provides two **smart pointer** types that manage dynamic objects. A smart pointer acts like a regular pointer with the important exception that it automatically deletes the object to which it points. The new library defines two kinds of smart pointers that differ in how they manage their underlying pointers: **shared_ptr**, which allows multiple pointers to refer to the same object, and **unique_ptr**, which "owns" the object to which it points. The library also defines a companion class named **weak_ptr** that is a weak reference to an object managed by a `shared_ptr`. All three are defined in the `memory` header.

12.1.1. The shared_ptr Class

Like `vectors`, smart pointers are templates (§ 3.3, p. 96). Therefore, when we create a smart pointer, we must supply additional information—in this case, the type to which the pointer can point. As with `vector`, we supply that type inside angle brackets that follow the name of the kind of smart pointer we are defining:

Click here to view code image

```
shared_ptr<string> p1;      // shared_ptr that can point at a string
shared_ptr<list<int>> p2; // shared_ptr that can point at a list of ints
```

A default initialized smart pointer holds a null pointer (§ 2.3.2, p. 53). In § 12.1.3 (p. 464), we'll cover additional ways to initialize a smart pointer.

We use a smart pointer in ways that are similar to using a pointer. Dereferencing a smart pointer returns the object to which the pointer points. When we use a smart pointer in a condition, the effect is to test whether the pointer is null:

Click here to view code image

```
// if p1 is not null, check whether it's the empty string
if (p1 && p1->empty())
    *p1 = "hi";   // if so, dereference p1 to assign a new value to that string
```

Table 12.1 (overleaf) lists operations common to `shared_ptr` and `unique_ptr`.

Those that are particular to `shared_ptr` are listed in Table 12.2 (p. 453).

Table 12.1. Operations Common to shared_ptr and unique_ptr

`shared_ptr<T> sp` `unique_ptr<T> up`	Null smart pointer that can point to objects of type `T`.
`p`	Use `p` as a condition; `true` if `p` points to an object.
`*p`	Dereference `p` to get the object to which `p` points.
`p->mem`	Synonym for `(*p).mem`.
`p.get()`	Returns the pointer in `p`. Use with caution; the object to which the returned pointer points will disappear when the smart pointer deletes it.
`swap(p, q)` `p.swap(q)`	Swaps the pointers in `p` and `q`.

Table 12.2. Operations Specific to shared_ptr

`make_shared<T>(args)`	Returns a `shared_ptr` pointing to a dynamically allocated object of type `T`. Uses *args* to initialize that object.
`shared_ptr<T> p(q)`	`p` is a copy of the `shared_ptr` `q`; increments the count in `q`. The pointer in `q` must be convertible to `T*` (§ 4.11.2, p. 161).
`p = q`	`p` and `q` are `shared_ptr`s holding pointers that can be converted to one another. Decrements `p`'s reference count and increments `q`'s count; deletes `p`'s existing memory if `p`'s count goes to 0.
`p.unique()`	Returns `true` if `p.use_count()` is one; `false` otherwise.
`p.use_count()`	Returns the number of objects sharing with `p`; may be a slow operation, intended primarily for debugging purposes.

The make_shared Function

The safest way to allocate and use dynamic memory is to call a library function named `make_shared`. This function allocates and initializes an object in dynamic memory and returns a `shared_ptr` that points to that object. Like the smart pointers, `make_shared` is defined in the `memory` header.

When we call `make_shared`, we must specify the type of object we want to create. We do so in the same way as we use a template class, by following the function name with a type enclosed in angle brackets:

Click here to view code image

```
// shared_ptr that points to an int with value 42
shared_ptr<int> p3 = make_shared<int>(42);
// p4 points to a string with value 9999999999
shared_ptr<string> p4 = make_shared<string>(10, '9');
```

```
// p5 points to an int that is value initialized (§ 3.3.1 (p. 98)) to 0
shared_ptr<int> p5 = make_shared<int>();
```

Like the sequential-container `emplace` members (§ 9.3.1, p. 345), `make_shared` uses its arguments to construct an object of the given type. For example, a call to `make_shared<string>` must pass argument(s) that match one of the `string` constructors. Calls to `make_shared<int>` can pass any value we can use to initialize an `int`. And so on. If we do not pass any arguments, then the object is value initialized (§ 3.3.1, p. 98).

Of course, ordinarily we use `auto` (§ 2.5.2, p. 68) to make it easier to define an object to hold the result of `make_shared`:

Click here to view code image

```
// p6 points to a dynamically allocated, empty vector<string>
auto p6 = make_shared<vector<string>>();
```

Copying and Assigning shared_ptrs

When we copy or assign a `shared_ptr`, each `shared_ptr` keeps track of how many other `shared_ptrs` point to the same object:

Click here to view code image

```
auto p = make_shared<int>(42); // object to which p points has one user
auto q(p); // p and q point to the same object
           // object to which p and q point has two users
```

We can think of a `shared_ptr` as if it has an associated counter, usually referred to as a **reference count**. Whenever we copy a `shared_ptr`, the count is incremented. For example, the counter associated with a `shared_ptr` is incremented when we use it to initialize another `shared_ptr`, when we use it as the right-hand operand of an assignment, or when we pass it to (§ 6.2.1, p. 209) or return it from a function by value (§ 6.3.2, p. 224). The counter is decremented when we assign a new value to the `shared_ptr` and when the `shared_ptr` itself is destroyed, such as when a local `shared_ptr` goes out of scope (§ 6.1.1, p. 204).

Once a `shared_ptr`'s counter goes to zero, the `shared_ptr` automatically frees the object that it manages:

Click here to view code image

```
auto r = make_shared<int>(42); // int to which r points has one user
r = q;  // assign to r, making it point to a different address
        // increase the use count for the object to which q points
        // reduce the use count of the object to which r had pointed
        // the object r had pointed to has no users; that object is automatically
freed
```

Here we allocate an `int` and store a pointer to that `int` in `r`. Next, we assign a new value to `r`. In this case, `r` is the only `shared_ptr` pointing to the one we previously allocated. That `int` is automatically freed as part of assigning `q` to `r`.

Note

It is up to the implementation whether to use a counter or another data structure to keep track of how many pointers share state. The key point is that the class keeps track of how many `shared_ptrs` point to the same object and automatically frees that object when appropriate.

shared_ptrs Automatically Destroy Their Objects ...

When the last `shared_ptr` pointing to an object is destroyed, the `shared_ptr` class automatically destroys the object to which that `shared_ptr` points. It does so through another special member function known as a **destructor**. Analogous to its constructors, each class has a destructor. Just as a constructor controls initialization, the destructor controls what happens when objects of that class type are destroyed.

Destructors generally free the resources that an object has allocated. For example, the `string` constructors (and other `string` members) allocate memory to hold the characters that compose the `string`. The `string` destructor frees that memory. Similarly, several `vector` operations allocate memory to hold the elements in the `vector`. The destructor for `vector` destroys those elements and frees the memory used for the elements.

The destructor for `shared_ptr` decrements the reference count of the object to which that `shared_ptr` points. If the count goes to zero, the `shared_ptr` destructor destroys the object to which the `shared_ptr` points and frees the memory used by that object.

...and Automatically Free the Associated Memory

The fact that the `shared_ptr` class automatically frees dynamic objects when they are no longer needed makes it fairly easy to use dynamic memory. For example, we might have a function that returns a `shared_ptr` to a dynamically allocated object of a type named `Foo` that can be initialized by an argument of type `T`:

Click here to view code image

```
// factory returns a shared_ptr pointing to a dynamically allocated object
shared_ptr<Foo> factory(T arg)
{
    // process arg as appropriate
```

```
    // shared_ptr will take care of deleting this memory
    return make_shared<Foo>(arg);
}
```

Because `factory` returns a `shared_ptr`, we can be sure that the object allocated by `factory` will be freed when appropriate. For example, the following function stores the `shared_ptr` returned by `factory` in a local variable:

Click here to view code image

```
void use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg);
    // use p
} // p goes out of scope; the memory to which p points is automatically freed
```

Because p is local to `use_factory`, it is destroyed when `use_factory` ends (§ 6.1.1, p. 204). When p is destroyed, its reference count is decremented and checked. In this case, p is the only object referring to the memory returned by `factory`. Because p is about to go away, the object to which p points will be destroyed and the memory in which that object resides will be freed.

The memory will not be freed if there is any other `shared_ptr` pointing to it:

Click here to view code image

```
shared_ptr<Foo> use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg);
    // use p
    return p;  // reference count is incremented when we return p
} // p goes out of scope; the memory to which p points is not freed
```

In this version, the `return` statement in `use_factory` returns a copy of p to its caller (§ 6.3.2, p. 224). Copying a `shared_ptr` adds to the reference count of that object. Now when p is destroyed, there will be another user for the memory to which p points. The `shared_ptr` class ensures that so long as there are any `shared_ptrs` attached to that memory, the memory itself will not be freed.

Because memory is not freed until the last `shared_ptr` goes away, it can be important to be sure that `shared_ptrs` don't stay around after they are no longer needed. The program will execute correctly but may waste memory if you neglect to destroy `shared_ptrs` that the program does not need. One way that `shared_ptrs` might stay around after you need them is if you put `shared_ptrs` in a container and subsequently reorder the container so that you don't need all the elements. You should be sure to `erase` `shared_ptr` elements once you no longer need those elements.

Note

> If you put `shared_ptrs` in a container, and you subsequently need to use some, but not all, of the elements, remember to `erase` the elements you no longer need.

Classes with Resources That Have Dynamic Lifetime

Programs tend to use dynamic memory for one of three purposes:

1. They don't know how many objects they'll need
2. They don't know the precise type of the objects they need
3. They want to share data between several objects

The container classes are an example of classes that use dynamic memory for the first purpose and we'll see examples of the second in Chapter 15. In this section, we'll define a class that uses dynamic memory in order to let several objects share the same underlying data.

So far, the classes we've used allocate resources that exist only as long as the corresponding objects. For example, each `vector` "owns" its own elements. When we copy a `vector`, the elements in the original `vector` and in the copy are separate from one another:

Click here to view code image

```
vector<string> v1; // empty vector
{ // new scope
    vector<string> v2 = {"a", "an", "the"};
    v1 = v2; // copies the elements from v2 into v1
} // v2 is destroyed, which destroys the elements in v2
  // v1 has three elements, which are copies of the ones originally in v2
```

The elements allocated by a `vector` exist only while the `vector` itself exists. When a `vector` is destroyed, the elements in the `vector` are also destroyed.

Some classes allocate resources with a lifetime that is independent of the original object. As an example, assume we want to define a class named `Blob` that will hold a collection of elements. Unlike the containers, we want `Blob` objects that are copies of one another to share the same elements. That is, when we copy a `Blob`, the original and the copy should refer to the same underlying elements.

In general, when two objects share the same underlying data, we can't unilaterally destroy the data when an object of that type goes away:

Click here to view code image

```
Blob<string> b1;     // empty Blob
{ // new scope
```

```
        Blob<string> b2 = {"a", "an", "the"};
        b1 = b2; // b1 and b2 share the same elements
    } // b2 is destroyed, but the elements in b2 must not be destroyed
      // b1 points to the elements originally created in b2
```

In this example, `b1` and `b2` share the same elements. When `b2` goes out of scope, those elements must stay around, because `b1` is still using them.

Note

One common reason to use dynamic memory is to allow multiple objects to share the same state.

Defining the StrBlob Class

Ultimately, we'll implement our `Blob` class as a template, but we won't learn how to do so until § 16.1.2 (p. 658). For now, we'll define a version of our class that can manage `strings`. As a result, we'll name this version of our class `StrBlob`.

The easiest way to implement a new collection type is to use one of the library containers to manage the elements. That way, we can let the library type manage the storage for the elements themselves. In this case, we'll use a `vector` to hold our elements.

However, we can't store the `vector` directly in a `Blob` object. Members of an object are destroyed when the object itself is destroyed. For example, assume that `b1` and `b2` are two `Blobs` that share the same `vector`. If that `vector` were stored in one of those `Blobs`—say, `b2`—then that `vector`, and therefore its elements, would no longer exist once `b2` goes out of scope. To ensure that the elements continue to exist, we'll store the `vector` in dynamic memory.

To implement the sharing we want, we'll give each `StrBlob` a `shared_ptr` to a dynamically allocated `vector`. That `shared_ptr` member will keep track of how many `StrBlobs` share the same `vector` and will delete the `vector` when the last `StrBlob` using that `vector` is destroyed.

We still need to decide what operations our class will provide. For now, we'll implement a small subset of the `vector` operations. We'll also change the operations that access elements (e.g., `front` and `back`): In our class, these operations will throw an exception if a user attempts to access an element that doesn't exist.

Our class will have a default constructor and a constructor that has a parameter of type `initializer_list<string>` (§ 6.2.6, p. 220). This constructor will take a braced list of initializers.

Click here to view code image

```
class StrBlob {
public:
    typedef std::vector<std::string>::size_type size_type;
    StrBlob();
    StrBlob(std::initializer_list<std::string> il);
    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }
    // add and remove elements
          void push_back(const std::string &t) {data-
>push_back(t);}
    void pop_back();
    // element access
    std::string& front();
    std::string& back();
private:
    std::shared_ptr<std::vector<std::string>> data;
    // throws msg if data[i] isn't valid
    void check(size_type i, const std::string &msg) const;
};
```

Inside the class we implemented the `size`, `empty`, and `push_back` members. These members forward their work through the `data` pointer to the underlying `vector`. For example, `size()` on a `StrBlob` calls `data->size()`, and so on.

StrBlob Constructors

Each constructor uses its constructor initializer list (§ 7.1.4, p. 265) to initialize its `data` member to point to a dynamically allocated `vector`. The default constructor allocates an empty `vector`:

Click here to view code image

```
StrBlob::StrBlob(): data(make_shared<vector<string>>()) { }
StrBlob::StrBlob(initializer_list<string> il):
                 data(make_shared<vector<string>>(il)) { }
```

The constructor that takes an `initializer_list` passes its parameter to the corresponding `vector` constructor (§ 2.2.1, p. 43). That constructor initializes the `vector`'s elements by copying the values in the list.

Element Access Members

The `pop_back`, `front`, and `back` operations access members in the `vector`. These operations must check that an element exists before attempting to access that element. Because several members need to do the same checking, we've given our class a `private` utility function named `check` that verifies that a given index is in range. In addition to an index, `check` takes a `string` argument that it will pass to the exception handler. The `string` describes what went wrong:

Click here to view code image

```
void StrBlob::check(size_type i, const string &msg) const
{
    if (i >= data->size())
        throw out_of_range(msg);
}
```

The `pop_back` and element access members first call `check`. If `check` succeeds, these members forward their work to the underlying `vector` operation:

Click here to view code image

```
string& StrBlob::front()
{
    // if the vector is empty, check will throw
    check(0, "front on empty StrBlob");
    return data->front();
}
string& StrBlob::back()
{
    check(0, "back on empty StrBlob");
    return data->back();
}
void StrBlob::pop_back()
{
    check(0, "pop_back on empty StrBlob");
    data->pop_back();
}
```

The `front` and `back` members should be overloaded on `const` (§ 7.3.2, p. 276). Defining those versions is left as an exercise.

Copying, Assigning, and Destroying StrBlobs

Like our `Sales_data` class, `StrBlob` uses the default versions of the operations that copy, assign, and destroy objects of its type (§ 7.1.5, p. 267). By default, these operations copy, assign, and destroy the data members of the class. Our `StrBlob` has only one data member, which is a `shared_ptr`. Therefore, when we copy, assign, or destroy a `StrBlob`, its `shared_ptr` member will be copied, assigned, or destroyed.

As we've seen, copying a `shared_ptr` increments its reference count; assigning one `shared_ptr` to another increments the count of the right-hand operand and decrements the count in the left-hand operand; and destroying a `shared_ptr` decrements the count. If the count in a `shared_ptr` goes to zero, the object to which that `shared_ptr` points is automatically destroyed. Thus, the `vector` allocated by the `StrBlob` constructors will be automatically destroyed when the last `StrBlob` pointing to that `vector` is destroyed.

Exercises Section 12.1.1

Exercise 12.1: How many elements do `b1` and `b2` have at the end of this code?

Click here to view code image

```
StrBlob b1;
{
    StrBlob b2 = {"a", "an", "the"};
    b1 = b2;
    b2.push_back("about");
}
```

Exercise 12.2: Write your own version of the `StrBlob` class including the `const` versions of `front` and `back`.

Exercise 12.3: Does this class need `const` versions of `push_back` and `pop_back`? If so, add them. If not, why aren't they needed?

Exercise 12.4: In our `check` function we didn't check whether `i` was greater than zero. Why is it okay to omit that check?

Exercise 12.5: We did not make the constructor that takes an `initializer_list` `explicit` (§ 7.5.4, p. 296). Discuss the pros and cons of this design choice.

12.1.2. Managing Memory Directly

The language itself defines two operators that allocate and free dynamic memory. The `new` operator allocates memory, and `delete` frees memory allocated by `new`.

For reasons that will become clear as we describe how these operators work, using these operators to manage memory is considerably more error-prone than using a smart pointer. Moreover, classes that do manage their own memory—unlike those that use smart pointers—cannot rely on the default definitions for the members that copy, assign, and destroy class objects (§ 7.1.4, p. 264). As a result, programs that use smart pointers are likely to be easier to write and debug.

Warning

Until you have read Chapter 13, your classes should allocate dynamic memory *only* if they use smart pointers to manage that memory.

Using `new` to Dynamically Allocate and Initialize Objects

Objects allocated on the free store are unnamed, so **`new`** offers no way to name the

objects that it allocates. Instead, new returns a pointer to the object it allocates:

Click here to view code image

```
int *pi = new int;          // pi points to a dynamically allocated,
                            // unnamed, uninitialized int
```

This new expression constructs an object of type int on the free store and returns a pointer to that object.

By default, dynamically allocated objects are default initialized (§ 2.2.1, p. 43), which means that objects of built-in or compound type have undefined value; objects of class type are initialized by their default constructor:

Click here to view code image

```
string *ps = new string;   // initialized to empty string
int *pi = new int;         // pi points to an uninitialized int
```

C++ 11

We can initialize a dynamically allocated object using direct initialization (§ 3.2.1, p. 84). We can use traditional construction (using parentheses), and under the new standard, we can also use list initialization (with curly braces):

Click here to view code image

```
int *pi = new int(1024); // object to which pi points has value 1024
string *ps = new string(10, '9');    // *ps is "9999999999"
// vector with ten elements with values from 0 to 9
vector<int> *pv = new vector<int>{0,1,2,3,4,5,6,7,8,9};
```

We can also value initialize (§ 3.3.1, p. 98) a dynamically allocated object by following the type name with a pair of empty parentheses:

Click here to view code image

```
string *ps1 = new string;   // default initialized to the empty string
string *ps = new string(); // value initialized to the empty string
int *pi1 = new int;         // default initialized; *pi1 is undefined
int *pi2 = new int();       // value initialized to 0; *pi2 is 0
```

For class types (such as string) that define their own constructors (§ 7.1.4, p. 262), requesting value initialization is of no consequence; regardless of form, the object is initialized by the default constructor. In the case of built-in types the difference is significant; a value-initialized object of built-in type has a well-defined value but a default-initialized object does not. Similarly, members of built-in type in classes that rely on the synthesized default constructor will also be uninitialized if those members are not initialized in the class body (§ 7.1.4, p. 263).

Best Practices

> For the same reasons as we usually initialize variables, it is also a good idea to initialize dynamically allocated objects.

C++ 11

When we provide an initializer inside parentheses, we can use `auto` (§ 2.5.2, p. 68) to deduce the type of the object we want to allocate from that initializer. However, because the compiler uses the initializer's type to deduce the type to allocate, we can use `auto` only with a single initializer inside parentheses:

Click here to view code image

```
auto p1 = new auto(obj);    // p points to an object of the type of obj
                            // that object is initialized from obj
auto p2 = new auto{a,b,c}; // error: must use parentheses for the initializer
```

The type of `p1` is a pointer to the `auto`-deduced type of `obj`. If `obj` is an `int`, then `p1` is `int*`; if `obj` is a `string`, then `p1` is a `string*`; and so on. The newly allocated object is initialized from the value of `obj`.

Dynamically Allocated const Objects

It is legal to use `new` to allocate `const` objects:

Click here to view code image

```
// allocate and initialize a const int
const int *pci = new const int(1024);
// allocate a default-initialized const empty string
const string *pcs = new const string;
```

Like any other `const`, a dynamically allocated `const` object must be initialized. A `const` dynamic object of a class type that defines a default constructor (§ 7.1.4, p. 263) may be initialized implicitly. Objects of other types must be explicitly initialized. Because the allocated object is `const`, the pointer returned by `new` is a pointer to `const` (§ 2.4.2, p. 62).

Memory Exhaustion

Although modern machines tend to have huge memory capacity, it is always possible that the free store will be exhausted. Once a program has used all of its available memory, `new` expressions will fail. By default, if `new` is unable to allocate the requested storage, it throws an exception of type `bad_alloc` (§ 5.6, p. 193). We can prevent `new` from throwing an exception by using a different form of `new`:

Click here to view code image

```
// if allocation fails, new returns a null pointer
int *p1 = new int; // if allocation fails, new throws std::bad_alloc
int *p2 = new (nothrow) int; // if allocation fails, new returns a null pointer
```

For reasons we'll explain in § 19.1.2 (p. 824) this form of `new` is referred to as **placement new**. A placement new expression lets us pass additional arguments to `new`. In this case, we pass an object named `nothrow` that is defined by the library. When we pass `nothrow` to `new`, we tell `new` that it must not throw an exception. If this form of `new` is unable to allocate the requested storage, it will return a null pointer. Both `bad_alloc` and `nothrow` are defined in the `new` header.

Freeing Dynamic Memory

In order to prevent memory exhaustion, we must return dynamically allocated memory to the system once we are finished using it. We return memory through a **delete expression**. A `delete` expression takes a pointer to the object we want to free:

Click here to view code image

```
delete p;       // p must point to a dynamically allocated object or be null
```

Like `new`, a `delete` expression performs two actions: It destroys the object to which its given pointer points, and it frees the corresponding memory.

Pointer Values and delete

The pointer we pass to `delete` must either point to dynamically allocated memory or be a null pointer (§ 2.3.2, p. 53). Deleting a pointer to memory that was not allocated by `new`, or deleting the same pointer value more than once, is undefined:

Click here to view code image

```
int i, *pi1 = &i, *pi2 = nullptr;
double *pd = new double(33), *pd2 = pd;
delete i;   // error: i is not a pointer
delete pi1; // undefined: pi1 refers to a local
delete pd;  // ok
delete pd2; // undefined: the memory pointed to by pd2 was already freed
delete pi2; // ok: it is always ok to delete a null pointer
```

The compiler will generate an error for the `delete` of `i` because it knows that `i` is not a pointer. The errors associated with executing `delete` on `pi1` and `pd2` are more insidious: In general, compilers cannot tell whether a pointer points to a statically or dynamically allocated object. Similarly, the compiler cannot tell whether memory addressed by a pointer has already been freed. Most compilers will accept these `delete` expressions, even though they are in error.

Although the value of a `const` object cannot be modified, the object itself can be destroyed. As with any other dynamic object, a `const` dynamic object is freed by executing `delete` on a pointer that points to that object:

Click here to view code image

```
const int *pci = new const int(1024);
delete pci;  // ok: deletes a const object
```

Dynamically Allocated Objects Exist until They Are Freed

As we saw in § 12.1.1 (p. 452), memory that is managed through a `shared_ptr` is automatically deleted when the last `shared_ptr` is destroyed. The same is not true for memory we manage using built-in pointers. A dynamic object managed through a built-in pointer exists until it is explicitly deleted.

Functions that return pointers (rather than smart pointers) to dynamic memory put a burden on their callers—the caller must remember to delete the memory:

Click here to view code image

```
// factory returns a pointer to a dynamically allocated object
Foo* factory(T arg)
{
    // process arg as appropriate
    return new Foo(arg); // caller is responsible for deleting this memory
}
```

Like our earlier `factory` function (§ 12.1.1, p. 453), this version of `factory` allocates an object but does not `delete` it. Callers of `factory` are responsible for freeing this memory when they no longer need the allocated object. Unfortunately, all too often the caller forgets to do so:

Click here to view code image

```
void use_factory(T arg)
{
    Foo *p = factory(arg);
    // use p but do not delete it
} // p goes out of scope, but the memory to which p points is not freed!
```

Here, our `use_factory` function calls `factory`, which allocates a new object of type `Foo`. When `use_factory` returns, the local variable `p` is destroyed. That variable is a built-in pointer, not a smart pointer.

Unlike class types, nothing happens when objects of built-in type are destroyed. In particular, when a pointer goes out of scope, nothing happens to the object to which the pointer points. If that pointer points to dynamic memory, that memory is not automatically freed.

Warning

Dynamic memory managed through built-in pointers (rather than smart pointers) exists until it is explicitly freed.

In this example, p was the only pointer to the memory allocated by `factory`. Once `use_factory` returns, the program has no way to free that memory. Depending on the logic of our overall program, we should fix this bug by remembering to free the memory inside `use_factory`:

Click here to view code image

```
void use_factory(T arg)
{
    Foo *p = factory(arg);
    // use p
    delete p;   // remember to free the memory now that we no longer need it
}
```

or, if other code in our system needs to use the object allocated by `use_factory`, we should change that function to return a pointer to the memory it allocated:

Click here to view code image

```
Foo* use_factory(T arg)
{
    Foo *p = factory(arg);
    // use p
    return p;   // caller must delete the memory
}
```

Caution: Managing Dynamic Memory Is Error-Prone

There are three common problems with using `new` and `delete` to manage dynamic memory:

1. Forgetting to `delete` memory. Neglecting to delete dynamic memory is known as a "memory leak," because the memory is never returned to the free store. Testing for memory leaks is difficult because they usually cannot be detected until the application is run for a long enough time to actually exhaust memory.
2. Using an object after it has been deleted. This error can sometimes be detected by making the pointer null after the delete.
3. Deleting the same memory twice. This error can happen when two pointers address the same dynamically allocated object. If `delete` is applied to one of the pointers, then the object's memory is returned to the free store. If we subsequently `delete` the second pointer, then the free store may be corrupted.

These kinds of errors are considerably easier to make than they are to find and fix.

Best Practices

You can avoid *all* of these problems by using smart pointers exclusively. The smart pointer will take care of deleting the memory *only* when there are no remaining smart pointers pointing to that memory.

Resetting the Value of a Pointer after a delete ...

When we `delete` a pointer, that pointer becomes invalid. Although the pointer is invalid, on many machines the pointer continues to hold the address of the (freed) dynamic memory. After the `delete`, the pointer becomes what is referred to as a **dangling pointer**. A dangling pointer is one that refers to memory that once held an object but no longer does so.

Dangling pointers have all the problems of uninitialized pointers (§ 2.3.2, p. 54). We can avoid the problems with dangling pointers by deleting the memory associated with a pointer just before the pointer itself goes out of scope. That way there is no chance to use the pointer after the memory associated with the pointer is freed. If we need to keep the pointer around, we can assign `nullptr` to the pointer after we use `delete`. Doing so makes it clear that the pointer points to no object.

...Provides Only Limited Protection

A fundamental problem with dynamic memory is that there can be several pointers that point to the same memory. Resetting the pointer we use to `delete` that memory lets us check that particular pointer but has no effect on any of the other pointers that still point at the (freed) memory. For example:

Click here to view code image

```
int *p(new int(42));   // p points to dynamic memory
auto q = p;            // p and q point to the same memory
delete p;     // invalidates both p and q
p = nullptr; // indicates that p is no longer bound to an object
```

Here both `p` and `q` point at the same dynamically allocated object. We `delete` that memory and set `p` to `nullptr`, indicating that the pointer no longer points to an object. However, resetting `p` has no effect on `q`, which became invalid when we deleted the memory to which `p` (and `q`!) pointed. In real systems, finding all the

pointers that point to the same memory is surprisingly difficult.

Exercises Section 12.1.2

Exercise 12.6: Write a function that returns a dynamically allocated `vector` of `int`s. Pass that `vector` to another function that reads the standard input to give values to the elements. Pass the `vector` to another function to print the values that were read. Remember to `delete` the `vector` at the appropriate time.

Exercise 12.7: Redo the previous exercise, this time using `shared_ptr`.

Exercise 12.8: Explain what if anything is wrong with the following function.

```
bool b() {
    int* p = new int;
    // …
    return p;
}
```

Exercise 12.9: Explain what happens in the following code:

Click here to view code image

```
int *q = new int(42), *r = new int(100);
r = q;
auto q2 = make_shared<int>(42), r2 = make_shared<int>(100);
r2 = q2;
```

12.1.3. Using shared_ptrs with new

As we've seen, if we do not initialize a smart pointer, it is initialized as a null pointer. As described in Table 12.3, we can also initialize a smart pointer from a pointer returned by `new`:

Click here to view code image

```
shared_ptr<double> p1; // shared_ptr that can point at a double
shared_ptr<int> p2(new int(42)); // p2 points to an int with value 42
```

Table 12.3. Other Ways to Define and Change shared_ptrs

`shared_ptr<T> p(q)`	p manages the object to which the built-in pointer q points; q must point to memory allocated by new and must be convertible to `T*`.
`shared_ptr<T> p(u)`	p assumes ownership from the `unique_ptr` u; makes u null.
`shared_ptr<T> p(q, d)`	p assumes ownership for the object to which the built-in pointer q points. q must be convertible to `T*` (§ 4.11.2, p. 161). p will use the callable object d (§ 10.3.2, p. 388) in place of `delete` to free q.
`shared_ptr<T> p(p2, d)`	p is a copy of the `shared_ptr` p2 as described in Table 12.2 except that p uses the callable object d in place of `delete`.

`p.reset()`
`p.reset(q)`
`p.reset(q, d)`
	If p is the only `shared_ptr` pointing at its object, `reset` frees p's existing object. If the optional built-in pointer q is passed, makes p point to q, otherwise makes p null. If d is supplied, will call d to free q otherwise uses `delete` to free q.

The smart pointer constructors that take pointers are `explicit` (§ 7.5.4, p. 296). Hence, we cannot implicitly convert a built-in pointer to a smart pointer; we must use the direct form of initialization (§ 3.2.1, p. 84) to initialize a smart pointer:

Click here to view code image

```
shared_ptr<int> p1 = new int(1024);   // error: must use direct initialization
shared_ptr<int> p2(new int(1024));    // ok: uses direct initialization
```

The initialization of `p1` implicitly asks the compiler to create a `shared_ptr` from the `int*` returned by `new`. Because we can't implicitly convert a pointer to a smart pointer, this initialization is an error. For the same reason, a function that returns a `shared_ptr` cannot implicitly convert a plain pointer in its return statement:

Click here to view code image

```
shared_ptr<int> clone(int p) {
    return new int(p); // error: implicit conversion to shared_ptr<int>
}
```

We must explicitly bind a `shared_ptr` to the pointer we want to return:

Click here to view code image

```
shared_ptr<int> clone(int p) {
    // ok: explicitly create a shared_ptr<int> from int*
    return shared_ptr<int>(new int(p));
}
```

By default, a pointer used to initialize a smart pointer must point to dynamic memory because, by default, smart pointers use `delete` to free the associated object. We can bind smart pointers to pointers to other kinds of resources. However, to do so, we must supply our own operation to use in place of `delete`. We'll see how to supply our own deletion code in § 12.1.4 (p. 468).

Don’t Mix Ordinary Pointers and Smart Pointers ...

A `shared_ptr` can coordinate destruction only with other `shared_ptrs` that are copies of itself. Indeed, this fact is one of the reasons we recommend using `make_shared` rather than `new`. That way, we bind a `shared_ptr` to the object at the same time that we allocate it. There is no way to inadvertently bind the same memory to more than one independently created `shared_ptr`.

Consider the following function that operates on a `shared_ptr`:

Click here to view code image

```
// ptr is created and initialized when process is called
void process(shared_ptr<int> ptr)
{
    // use ptr
} // ptr goes out of scope and is destroyed
```

The parameter to `process` is passed by value, so the argument to `process` is copied into `ptr`. Copying a `shared_ptr` increments its reference count. Thus, inside `process` the count is at least 2. When `process` completes, the reference count of `ptr` is decremented but cannot go to zero. Therefore, when the local variable `ptr` is destroyed, the memory to which `ptr` points will not be deleted.

The right way to use this function is to pass it a `shared_ptr`:

Click here to view code image

```
shared_ptr<int> p(new int(42)); // reference count is 1
process(p); // copying p increments its count; in process the reference count is 2
int i = *p; // ok: reference count is 1
```

Although we cannot pass a built-in pointer to `process`, we can pass `process` a (temporary) `shared_ptr` that we explicitly construct from a built-in pointer. However, doing so is likely to be an error:

Click here to view code image

```
int *x(new int(1024)); // dangerous: x is a plain pointer, not a smart pointer
process(x);   // error: cannot convert int* to shared_ptr<int>
process(shared_ptr<int>(x)); // legal, but the memory will be deleted!
int j = *x;   // undefined: x is a dangling pointer!
```

In this call, we passed a temporary `shared_ptr` to `process`. That temporary is destroyed when the expression in which the call appears finishes. Destroying the

temporary decrements the reference count, which goes to zero. The memory to which the temporary points is freed when the temporary is destroyed.

But `x` continues to point to that (freed) memory; `x` is now a dangling pointer. Attempting to use the value of `x` is undefined.

When we bind a `shared_ptr` to a plain pointer, we give responsibility for that memory to that `shared_ptr`. Once we give `shared_ptr` responsibility for a pointer, we should no longer use a built-in pointer to access the memory to which the `shared_ptr` now points.

> **Warning**
>
> It is dangerous to use a built-in pointer to access an object owned by a smart pointer, because we may not know when that object is destroyed.

...and Don't Use get to Initialize or Assign Another Smart Pointer

The smart pointer types define a function named `get` (described in Table 12.1 (p. 452)) that returns a built-in pointer to the object that the smart pointer is managing. This function is intended for cases when we need to pass a built-in pointer to code that can't use a smart pointer. The code that uses the return from `get` must not `delete` that pointer.

Although the compiler will not complain, it is an error to bind another smart pointer to the pointer returned by `get`:

Click here to view code image

```
shared_ptr<int> p(new int(42)); // reference count is 1
int *q = p.get();   // ok: but don't use q in any way that might delete its pointer
{ // new block
// undefined: two independent shared_ptrs point to the same memory
shared_ptr<int>(q);
} // block ends, q is destroyed, and the memory to which q points is freed
int foo = *p; // undefined; the memory to which p points was freed
```

In this case, both p and q point to the same memory. Because they were created independently from each other, each has a reference count of 1. When the block in which q was defined ends, q is destroyed. Destroying q frees the memory to which q points. That makes p into a dangling pointer, meaning that what happens when we attempt to use p is undefined. Moreover, when p is destroyed, the pointer to that memory will be `deleted` a second time.

Warning

Use `get` only to pass access to the pointer to code that you know will not `delete` the pointer. In particular, never use `get` to initialize or assign to another smart pointer.

Other shared_ptr Operations

The `shared_ptr` class gives us a few other operations, which are listed in Table 12.2 (p. 453) and Table 12.3 (on the previous page). We can use `reset` to assign a new pointer to a `shared_ptr`:

Click here to view code image

```
p = new int(1024);          // error: cannot assign a pointer to a shared_ptr
p.reset(new int(1024));     // ok: p points to a new object
```

Like assignment, `reset` updates the reference counts and, if appropriate, deletes the object to which `p` points. The `reset` member is often used together with `unique` to control changes to the object shared among several `shared_ptrs`. Before changing the underlying object, we check whether we're the only user. If not, we make a new copy before making the change:

Click here to view code image

```
if (!p.unique())
    p.reset(new string(*p)); // we aren't alone; allocate a new copy
*p += newVal; // now that we know we're the only pointer, okay to change this object
```

Exercises Section 12.1.3

Exercise 12.10: Explain whether the following call to the `process` function defined on page 464 is correct. If not, how would you correct the call?

Click here to view code image

```
shared_ptr<int> p(new int(42));
process(shared_ptr<int>(p));
```

Exercise 12.11: What would happen if we called `process` as follows?

Click here to view code image

```
process(shared_ptr<int>(p.get()));
```

Exercise 12.12: Using the declarations of `p` and `sp` explain each of the

following calls to `process`. If the call is legal, explain what it does. If the call is illegal, explain why:

```
auto p = new int();
auto sp = make_shared<int>();
```

(a) `process(sp);`

(b) `process(new int());`

(c) `process(p);`

(d) `process(shared_ptr<int>(p));`

Exercise 12.13: What happens if we execute the following code?

Click here to view code image

```
auto sp = make_shared<int>();
auto p = sp.get();
delete p;
```

12.1.4. Smart Pointers and Exceptions

In § 5.6.2 (p. 196) we noted that programs that use exception handling to continue processing after an exception occurs need to ensure that resources are properly freed if an exception occurs. One easy way to make sure resources are freed is to use smart pointers.

When we use a smart pointer, the smart pointer class ensures that memory is freed when it is no longer needed even if the block is exited prematurely:

Click here to view code image

```
void f()
{
    shared_ptr<int> sp(new int(42)); // allocate a new object
    // code that throws an exception that is not caught inside f
} // shared_ptr freed automatically when the function ends
```

When a function is exited, whether through normal processing or due to an exception, all the local objects are destroyed. In this case, `sp` is a `shared_ptr`, so destroying `sp` checks its reference count. Here, `sp` is the only pointer to the memory it manages; that memory will be freed as part of destroying `sp`.

In contrast, memory that we manage directly is not automatically freed when an exception occurs. If we use built-in pointers to manage memory and an exception occurs after a `new` but before the corresponding `delete`, then that memory won't be freed:

Click here to view code image

```
void f()
{
    int *ip = new int(42);       // dynamically allocate a new object
    // code that throws an exception that is not caught inside f
    delete ip;                   // free the memory before exiting
}
```

If an exception happens between the `new` and the `delete`, and is not caught inside `f`, then this memory can never be freed. There is no pointer to this memory outside the function `f`. Thus, there is no way to free this memory.

Smart Pointers and Dumb Classes

Many C++ classes, including all the library classes, define destructors (§ 12.1.1, p. 452) that take care of cleaning up the resources used by that object. However, not all classes are so well behaved. In particular, classes that are designed to be used by both C and C++ generally require the user to specifically free any resources that are used.

Classes that allocate resources—and that do not define destructors to free those resources—can be subject to the same kind of errors that arise when we use dynamic memory. It is easy to forget to release the resource. Similarly, if an exception happens between when the resource is allocated and when it is freed, the program will leak that resource.

We can often use the same kinds of techniques we use to manage dynamic memory to manage classes that do not have well-behaved destructors. For example, imagine we're using a network library that is used by both C and C++. Programs that use this library might contain code such as

Click here to view code image

```
struct destination;  // represents what we are connecting to
struct connection;   // information needed to use the connection
connection connect(destination*);  // open the connection
void disconnect(connection);       // close the given connection
void f(destination &d /* other parameters */)
{
    // get a connection; must remember to close it when done
    connection c = connect(&d);
    // use the connection
    // if we forget to call disconnect before exiting f, there will be no way to close
c
}
```

If `connection` had a destructor, that destructor would automatically close the connection when `f` completes. However, `connection` does not have a destructor. This problem is nearly identical to our previous program that used a `shared_ptr` to avoid memory leaks. It turns out that we can also use a `shared_ptr` to ensure that the `connection` is properly closed.

Using Our Own Deletion Code

By default, `shared_ptrs` assume that they point to dynamic memory. Hence, by default, when a `shared_ptr` is destroyed, it executes `delete` on the pointer it holds. To use a `shared_ptr` to manage a `connection`, we must first define a function to use in place of `delete`. It must be possible to call this **deleter** function with the pointer stored inside the `shared_ptr`. In this case, our deleter must take a single argument of type `connection*`:

Click here to view code image

```
void end_connection(connection *p) { disconnect(*p); }
```

When we create a `shared_ptr`, we can pass an optional argument that points to a deleter function (§ 6.7, p. 247):

Click here to view code image

```
void f(destination &d /* other parameters */)
{
    connection c = connect(&d);
    shared_ptr<connection> p(&c, end_connection);
    // use the connection
    // when f exits, even if by an exception, the connection will be properly closed
}
```

When `p` is destroyed, it won't execute `delete` on its stored pointer. Instead, `p` will call `end_connection` on that pointer. In turn, `end_connection` will call `disconnect`, thus ensuring that the connection is closed. If `f` exits normally, then `p` will be destroyed as part of the return. Moreover, `p` will also be destroyed, and the connection will be closed, if an exception occurs.

Caution: Smart Pointer Pitfalls

Smart pointers can provide safety and convenience for handling dynamically allocated memory only when they are used properly. To use smart pointers correctly, we must adhere to a set of conventions:

- Don't use the same built-in pointer value to initialize (or `reset`) more than one smart pointer.
- Don't `delete` the pointer returned from `get()`.

- Don't use `get()` to initialize or `reset` another smart pointer.
- If you use a pointer returned by `get()`, remember that the pointer will become invalid when the last corresponding smart pointer goes away.
- If you use a smart pointer to manage a resource other than memory allocated by `new`, remember to pass a deleter (§ 12.1.4, p. 468, and § 12.1.5, p. 471).

Exercises Section 12.1.4

Exercise 12.14: Write your own version of a function that uses a `shared_ptr` to manage a `connection`.

Exercise 12.15: Rewrite the first exercise to use a lambda (§ 10.3.2, p. 388) in place of the `end_connection` function.

12.1.5. unique_ptr

C++ 11

A `unique_ptr` "owns" the object to which it points. Unlike `shared_ptr`, only one `unique_ptr` at a time can point to a given object. The object to which a `unique_ptr` points is destroyed when the `unique_ptr` is destroyed. Table 12.4 lists the operations specific to `unique_ptrs`. The operations common to both were covered in Table 12.1 (p. 452).

Table 12.4. unique_ptr Operations (See Also Table 12.1 (p. 452))

`unique_ptr<T> u1` `unique_ptr<T, D> u2`	Null `unique_ptrs` that can point to objects of type `T`. `u1` will use `delete` to free its pointer; `u2` will use a callable object of type `D` to free its pointer.
`unique_ptr<T, D> u(d)`	Null `unique_ptr` that point to objects of type `T` that uses `d`, which must be an object of type `D` in place of `delete`.
`u = nullptr`	Deletes the object to which `u` points; makes `u` null.
`u.release()`	Relinquishes control of the pointer `u` had held; returns the pointer `u` had held and makes `u` null.

`u.reset()`
`u.reset(q)`
`u.reset(nullptr)`
	Deletes the object to which `u` points;
	If the built-in pointer `q` is supplied, makes `u` point to that object.
	Otherwise makes `u` null.

Unlike `shared_ptr`, there is no library function comparable to `make_shared` that returns a `unique_ptr`. Instead, when we define a `unique_ptr`, we bind it to a pointer returned by `new`. As with `shared_ptrs`, we must use the direct form of initialization:

Click here to view code image

```
unique_ptr<double> p1;  // unique_ptr that can point at a double
unique_ptr<int> p2(new int(42)); // p2 points to int with value 42
```

Because a `unique_ptr` owns the object to which it points, `unique_ptr` does not support ordinary copy or assignment:

Click here to view code image

```
unique_ptr<string> p1(new string("Stegosaurus"));
unique_ptr<string> p2(p1);   // error: no copy for unique_ptr
unique_ptr<string> p3;
p3 = p2;                     // error: no assign for unique_ptr
```

Although we can't copy or assign a `unique_ptr`, we can transfer ownership from one (`nonconst`) `unique_ptr` to another by calling `release` or `reset`:

Click here to view code image

```
// transfers ownership from p1 (which points to the string Stegosaurus) to p2
unique_ptr<string> p2(p1.release()); // release makes p1 null
unique_ptr<string> p3(new string("Trex"));
// transfers ownership from p3 to p2
p2.reset(p3.release()); // reset deletes the memory to which p2 had pointed
```

The `release` member returns the pointer currently stored in the `unique_ptr` and makes that `unique_ptr` null. Thus, `p2` is initialized from the pointer value that had been stored in `p1` and `p1` becomes null.

The `reset` member takes an optional pointer and repositions the `unique_ptr` to point to the given pointer. If the `unique_ptr` is not null, then the object to which the `unique_ptr` had pointed is deleted. The call to `reset` on `p2`, therefore, frees the memory used by the `string` initialized from `"Stegosaurus"`, transfers `p3`'s pointer to `p2`, and makes `p3` null.

Calling `release` breaks the connection between a `unique_ptr` and the object it had been managing. Often the pointer returned by `release` is used to initialize or assign another smart pointer. In that case, responsibility for managing the memory is simply transferred from one smart pointer to another. However, if we do not use another smart pointer to hold the pointer returned from `release`, our program takes over responsibility for freeing that resource:

Click here to view code image

```
p2.release(); // WRONG: p2 won't free the memory and we've lost the pointer
auto p = p2.release(); // ok, but we must remember to delete(p)
```

Passing and Returning unique_ptrs

There is one exception to the rule that we cannot copy a `unique_ptr`: We can copy or assign a `unique_ptr` that is about to be destroyed. The most common example is when we return a `unique_ptr` from a function:

Click here to view code image

```
unique_ptr<int> clone(int p) {
    // ok: explicitly create a unique_ptr<int> from int*
    return unique_ptr<int>(new int(p));
}
```

Alternatively, we can also return a copy of a local object:

Click here to view code image

```
unique_ptr<int> clone(int p) {
    unique_ptr<int> ret(new int (p));
    // . . .
    return ret;
}
```

In both cases, the compiler knows that the object being returned is about to be destroyed. In such cases, the compiler does a special kind of “copy” which we’ll discuss in § 13.6.2 (p. 534).

> **Backward Compatibility:** ***auto_ptr***
>
> Earlier versions of the library included a class named `auto_ptr` that had some, but not all, of the properties of `unique_ptr`. In particular, it was not possible to store an `auto_ptr` in a container, nor could we return one from a function.
>
> Although `auto_ptr` is still part of the standard library, programs should use `unique_ptr` instead.

Passing a Deleter to unique_ptr

Like `shared_ptr`, by default, `unique_ptr` uses `delete` to free the object to which a `unique_ptr` points. As with `shared_ptr`, we can override the default deleter in a `unique_ptr` (§ 12.1.4, p. 468). However, for reasons we’ll describe in § 16.1.6 (p. 676), the way `unique_ptr` manages its deleter is differs from the way `shared_ptr` does.

Overridding the deleter in a `unique_ptr` affects the `unique_ptr` type as well as how we construct (or `reset`) objects of that type. Similar to overriding the comparison operation of an associative container (§ 11.2.2, p. 425), we must supply the deleter type inside the angle brackets along with the type to which the

`unique_ptr` can point. We supply a callable object of the specified type when we create or `reset` an object of this type:

Click here to view code image

```
// p points to an object of type objT and uses an object of type delT to free that object
// it will call an object named fcn of type delT
unique_ptr<objT, delT> p (new objT, fcn);
```

As a somewhat more concrete example, we'll rewrite our connection program to use a `unique_ptr` in place of a `shared_ptr` as follows:

Click here to view code image

```
void f(destination &d /* other needed parameters */)
{
    connection c = connect(&d);  // open the connection
    // when p is destroyed, the connection will be closed
    unique_ptr<connection, decltype(end_connection)*>
        p(&c, end_connection);
    // use the connection
    // when f exits, even if by an exception, the connection will be properly closed
}
```

Here we use `decltype` (§ 2.5.3, p. 70) to specify the function pointer type. Because `decltype(end_connection)` returns a function type, we must remember to add a `*` to indicate that we're using a pointer to that type (§ 6.7, p. 250).

Exercises Section 12.1.5

Exercise 12.16: Compilers don't always give easy-to-understand error messages if we attempt to copy or assign a `unique_ptr`. Write a program that contains these errors to see how your compiler diagnoses them.

Exercise 12.17: Which of the following `unique_ptr` declarations are illegal or likely to result in subsequent program error? Explain what the problem is with each one.

Click here to view code image

```
int ix = 1024, *pi = &ix, *pi2 = new int(2048);
typedef unique_ptr<int> IntP;
```

(a) `IntP p0(ix);`

(b) `IntP p1(pi);`

(c) `IntP p2(pi2);`

(d) `IntP p3(&ix);`

(e) `IntP p4(new int(2048));`

(f) `IntP p5(p2.get());`

Exercise 12.18: Why doesn't `shared_ptr` have a `release` member?

12.1.6. weak_ptr

A `weak_ptr` (Table 12.5) is a smart pointer that does not control the lifetime of the object to which it points. Instead, a `weak_ptr` points to an object that is managed by a `shared_ptr`. Binding a `weak_ptr` to a `shared_ptr` does not change the reference count of that `shared_ptr`. Once the last `shared_ptr` pointing to the object goes away, the object itself will be deleted. That object will be deleted even if there are `weak_ptrs` pointing to it—hence the name `weak_ptr`, which captures the idea that a `weak_ptr` shares its object "weakly."

Table 12.5. weak_ptrs

`weak_ptr<T> w`	Null `weak_ptr` that can point at objects of type `T`.
`weak_ptr<T> w(sp)`	`weak_ptr` that points to the same object as the `shared_ptr sp`. `T` must be convertible to the type to which `sp` points.
`w = p`	`p` can be a `shared_ptr` or a `weak_ptr`. After the assignment `w` shares ownership with `p`.
`w.reset()`	Makes `w` null.
`w.use_count()`	The number of `shared_ptrs` that share ownership with `w`.
`w.expired()`	Returns `true` if `w.use_count()` is zero, `false` otherwise.
`w.lock()`	If `expired` is `true`, returns a null `shared_ptr`; otherwise returns a `shared_ptr` to the object to which `w` points.

When we create a `weak_ptr`, we initialize it from a `shared_ptr`:

Click here to view code image

```
auto p = make_shared<int>(42);
weak_ptr<int> wp(p);   // wp weakly shares with p; use count in p is unchanged
```

Here both `wp` and `p` point to the same object. Because the sharing is weak, creating `wp` doesn't change the reference count of `p`; it is possible that the object to which `wp` points might be deleted.

Because the object might no longer exist, we cannot use a `weak_ptr` to access its object directly. To access that object, we must call `lock`. The `lock` function checks whether the object to which the `weak_ptr` points still exists. If so, `lock` returns a `shared_ptr` to the shared object. As with any other `shared_ptr`, we are

guaranteed that the underlying object to which that `shared_ptr` points continues to exist at least as long as that `shared_ptr` exists. For example:

Click here to view code image

```
if (shared_ptr<int> np = wp.lock()) { // true if np is not null
    // inside the if, np shares its object with p
}
```

Here we enter the body of the `if` only if the call to `lock` succeeds. Inside the `if`, it is safe to use `np` to access that object.

Checked Pointer Class

As an illustration of when a `weak_ptr` is useful, we'll define a companion pointer class for our `StrBlob` class. Our pointer class, which we'll name `StrBlobPtr`, will store a `weak_ptr` to the `data` member of the `StrBlob` from which it was initialized. By using a `weak_ptr`, we don't affect the lifetime of the `vector` to which a given `StrBlob` points. However, we can prevent the user from attempting to access a `vector` that no longer exists.

`StrBlobPtr` will have two data members: `wptr`, which is either null or points to a `vector` in a `StrBlob`; and `curr`, which is the index of the element that this object currently denotes. Like its companion `StrBlob` class, our pointer class has a `check` member to verify that it is safe to dereference the `StrBlobPtr`:

Click here to view code image

```
// StrBlobPtr throws an exception on attempts to access a nonexistent element
class StrBlobPtr {
public:
    StrBlobPtr(): curr(0) { }
    StrBlobPtr(StrBlob &a, size_t sz = 0):
              wptr(a.data), curr(sz) { }
    std::string& deref() const;
    StrBlobPtr& incr();        // prefix version
private:
    // check returns a shared_ptr to the vector if the check succeeds
    std::shared_ptr<std::vector<std::string>>
        check(std::size_t, const std::string&) const;
    // store a weak_ptr, which means the underlying vector might be destroyed
    std::weak_ptr<std::vector<std::string>> wptr;
    std::size_t curr;       // current position within the array
};
```

The default constructor generates a null `StrBlobPtr`. Its constructor initializer list (§ 7.1.4, p. 265) explicitly initializes `curr` to zero and implicitly initializes `wptr` as a null `weak_ptr`. The second constructor takes a reference to `StrBlob` and an optional index value. This constructor initializes `wptr` to point to the `vector` in the `shared ptr` of the given `StrBlob` object and initializes `curr` to the value of `sz`.

We use a default argument (§ 6.5.1, p. 236) to initialize `curr` to denote the first element by default. As we'll see, the `sz` parameter will be used by the `end` member of `StrBlob`.

It is worth noting that we cannot bind a `StrBlobPtr` to a `const StrBlob` object. This restriction follows from the fact that the constructor takes a reference to a nonconst object of type `StrBlob`.

The `check` member of `StrBlobPtr` differs from the one in `StrBlob` because it must check whether the `vector` to which it points is still around:

Click here to view code image

```
std::shared_ptr<std::vector<std::string>>
StrBlobPtr::check(std::size_t i, const std::string &msg)
const
{
    auto ret = wptr.lock();    // is the vector still around?
    if (!ret)
        throw std::runtime_error("unbound StrBlobPtr");
    if (i >= ret->size())
        throw std::out_of_range(msg);
    return ret; // otherwise, return a shared_ptr to the vector
}
```

Because a `weak_ptr` does not participate in the reference count of its corresponding `shared_ptr`, the `vector` to which this `StrBlobPtr` points might have been deleted. If the `vector` is gone, `lock` will return a null pointer. In this case, any reference to the `vector` will fail, so we throw an exception. Otherwise, `check` verifies its given index. If that value is okay, `check` returns the `shared_ptr` it obtained from `lock`.

Pointer Operations

We'll learn how to define our own operators in Chapter 14. For now, we've defined functions named `deref` and `incr` to dereference and increment the `StrBlobPtr`, respectively.

The `deref` member calls `check` to verify that it is safe to use the `vector` and that `curr` is in range:

Click here to view code image

```
std::string& StrBlobPtr::deref() const
{
    auto p = check(curr, "dereference past end");
    return (*p)[curr];  // (*p) is the vector to which this object points
}
```

If `check` succeeds, `p` is a `shared_ptr` to the `vector` to which this `StrBlobPtr` points. The expression `(*p)[curr]` dereferences that `shared_ptr` to get the

vector and uses the subscript operator to fetch and return the element at curr.

The incr member also calls check:

Click here to view code image

```
// prefix: return a reference to the incremented object
StrBlobPtr& StrBlobPtr::incr()
{
    // if curr already points past the end of the container, can't increment it
    check(curr, "increment past end of StrBlobPtr");
    ++curr;         // advance the current state
    return *this;
}
```

Of course, in order to access the data member, our pointer class will have to be a friend of StrBlob (§ 7.3.4, p. 279). We'll also give our StrBlob class begin and end operations that return a StrBlobPtr pointing to itself:

Click here to view code image

```
// forward declaration needed for friend declaration in StrBlob
class StrBlobPtr;
class StrBlob {
    friend class StrBlobPtr;
    // other members as in § 12.1.1 (p. 456)
    // return StrBlobPtr to the first and one past the last elements
    StrBlobPtr begin() { return StrBlobPtr(*this); }
    StrBlobPtr end()
        { auto ret = StrBlobPtr(*this, data->size());
          return ret; }
};
```

Exercises Section 12.1.6

Exercise 12.19: Define your own version of StrBlobPtr and update your StrBlob class with the appropriate friend declaration and begin and end members.

Exercise 12.20: Write a program that reads an input file a line at a time into a StrBlob and uses a StrBlobPtr to print each element in that StrBlob.

Exercise 12.21: We could have written StrBlobPtr's deref member as follows:

Click here to view code image

```
std::string& deref() const
{ return (*check(curr, "dereference past end"))[curr]; }
```

Which version do you think is better and why?

Exercise 12.22: What changes would need to be made to `StrBlobPtr` to create a class that can be used with a `const StrBlob`? Define a class named `ConstStrBlobPtr` that can point to a `const StrBlob`.

12.2. Dynamic Arrays

The `new` and `delete` operators allocate objects one at a time. Some applications, need the ability to allocate storage for many objects at once. For example, `vectors` and `strings` store their elements in contiguous memory and must allocate several elements at once whenever the container has to be reallocated (§ 9.4, p. 355).

To support such usage, the language and library provide two ways to allocate an array of objects at once. The language defines a second kind of `new` expression that allocates and initializes an array of objects. The library includes a template class named `allocator` that lets us separate allocation from initialization. For reasons we'll explain in § 12.2.2 (p. 481), using an `allocator` generally provides better performance and more flexible memory management.

Many, perhaps even most, applications have no direct need for dynamic arrays. When an application needs a varying number of objects, it is almost always easier, faster, and safer to do as we did with `StrBlob`: use a `vector` (or other library container). For reasons we'll explain in § 13.6 (p. 531), the advantages of using a library container are even more pronounced under the new standard. Libraries that support the new standard tend to be dramatically faster than previous releases.

Best Practices

Most applications should use a library container rather than dynamically allocated arrays. Using a container is easier, less likely to contain memory-management bugs, *and* is likely to give better performance.

As we've seen, classes that use the containers can use the default versions of the operations for copy, assignment, and destruction (§ 7.1.5, p. 267). Classes that allocate dynamic arrays must define their own versions of these operations to manage the associated memory when objects are copied, assigned, and destroyed.

Warning

Do not allocate dynamic arrays in code inside classes until you have read Chapter 13.

12.2.1. new and Arrays

We ask new to allocate an array of objects by specifying the number of objects to allocate in a pair of square brackets after a type name. In this case, new allocates the requested number of objects and (assuming the allocation succeeds) returns a pointer to the first one:

Click here to view code image

```
// call get_size to determine how many ints to allocate
int *pia = new int[get_size()]; // pia points to the first of these ints
```

The size inside the brackets must have integral type but need not be a constant.

We can also allocate an array by using a type alias (§ 2.5.1, p. 67) to represent an array type. In this case, we omit the brackets:

Click here to view code image

```
typedef int arrT[42]; // arrT names the type array of 42 ints
int *p = new arrT;    // allocates an array of 42 ints; p points to the first one
```

Here, new allocates an array of ints and returns a pointer to the first one. Even though there are no brackets in our code, the compiler executes this expression using new[]. That is, the compiler executes this expression as if we had written

```
int *p = new int[42];
```

Allocating an Array Yields a Pointer to the Element Type

Although it is common to refer to memory allocated by new T[] as a "dynamic array," this usage is somewhat misleading. When we use new to allocate an array, we do not get an object with an array type. Instead, we get a pointer to the element type of the array. Even if we use a type alias to define an array type, new does not allocate an object of array type. In this case, the fact that we're allocating an array is not even visible; there is no [*num*]. Even so, new returns a pointer to the element type.

Because the allocated memory does not have an array type, we cannot call begin or end (§ 3.5.3, p. 118) on a dynamic array. These functions use the array dimension (which is part of an array's type) to return pointers to the first and one past the last elements, respectively. For the same reasons, we also cannot use a range for to process the elements in a (so-called) dynamic array.

Warning

It is important to remember that what we call a dynamic array does not have an array type.

Initializing an Array of Dynamically Allocated Objects

By default, objects allocated by `new`—whether allocated as a single object or in an array—are default initialized. We can value initialize (§ 3.3.1, p. 98) the elements in an array by following the size with an empty pair of parentheses.

Click here to view code image

```
int *pia = new int[10];            // block of ten uninitialized ints
int *pia2 = new int[10]();         // block of ten ints value initialized to
0
string *psa = new string[10];      // block of ten empty strings
string *psa2 = new string[10](); // block of ten empty strings
```

Under the new standard, we can also provide a braced list of element initializers:

Click here to view code image

```
// block of ten ints each initialized from the corresponding initializer
int *pia3 = new int[10]{0,1,2,3,4,5,6,7,8,9};
// block of ten strings; the first four are initialized from the given initializers
// remaining elements are value initialized
string *psa3 = new string[10]{"a", "an", "the",
string(3,'x')};
```

As when we list initialize an object of built-in array type (§ 3.5.1, p. 114), the initializers are used to initialize the first elements in the array. If there are fewer initializers than elements, the remaining elements are value initialized. If there are more initializers than the given size, then the `new` expression fails and no storage is allocated. In this case, `new` throws an exception of type `bad_array_new_length`. Like `bad_alloc`, this type is defined in the `new` header.

Although we can use empty parentheses to value initialize the elements of an array, we cannot supply an element initializer inside the parentheses. The fact that we cannot supply an initial value inside the parentheses means that we cannot use `auto` to allocate an array (§ 12.1.2, p. 459).

It Is Legal to Dynamically Allocate an Empty Array

We can use an arbitrary expression to determine the number of objects to allocate:

Click here to view code image

```
size_t n = get_size(); // get_size returns the number of elements needed
int* p = new int[n];   // allocate an array to hold the elements
for (int* q = p; q != p + n; ++q)
     /* process the array */ ;
```

An interesting question arises: What happens if `get_size` returns 0? The answer is that our code works fine. Calling `new[n]` with `n` equal to 0 is legal even though we cannot create an array variable of size 0:

Click here to view code image

```
char arr[0];              // error: cannot define a zero-length array
char *cp = new char[0]; // ok: but cp can't be dereferenced
```

When we use `new` to allocate an array of size zero, `new` returns a valid, nonzero pointer. That pointer is guaranteed to be distinct from any other pointer returned by `new`. This pointer acts as the off-the-end pointer (§ 3.5.3, p. 119) for a zero-element array. We can use this pointer in ways that we use an off-the-end iterator. The pointer can be compared as in the loop above. We can add zero to (or subtract zero from) such a pointer and can subtract the pointer from itself, yielding zero. The pointer cannot be dereferenced—after all, it points to no element.

In our hypothetical loop, if `get_size` returns 0, then `n` is also 0. The call to `new` will allocate zero objects. The condition in the `for` will fail (`p` is equal to `q + n` because `n` is 0). Thus, the loop body is not executed.

Freeing Dynamic Arrays

To free a dynamic array, we use a special form of `delete` that includes an empty pair of square brackets:

Click here to view code image

```
delete p;      // p must point to a dynamically allocated object or be null
delete [] pa; // pa must point to a dynamically allocated array or be null
```

The second statement destroys the elements in the array to which `pa` points and frees the corresponding memory. Elements in an array are destroyed in reverse order. That is, the last element is destroyed first, then the second to last, and so on.

When we `delete` a pointer to an array, the empty bracket pair is essential: It indicates to the compiler that the pointer addresses the first element of an array of objects. If we omit the brackets when we `delete` a pointer to an array (or provide them when we `delete` a pointer to an object), the behavior is undefined.

Recall that when we use a type alias that defines an array type, we can allocate an

array without using [] with new. Even so, we must use brackets when we delete a pointer to that array:

Click here to view code image

```
typedef int arrT[42];    // arrT names the type array of 42 ints
int *p = new arrT;       // allocates an array of 42 ints; p points to the first one
delete [] p;             // brackets are necessary because we allocated an array
```

Despite appearances, p points to the first element of an array of objects, not to a single object of type arrT. Thus, we must use [] when we delete p.

> **Warning**
>
> The compiler is unlikely to warn us if we forget the brackets when we delete a pointer to an array or if we use them when we delete a pointer to an object. Instead, our program is apt to misbehave without warning during execution.

Smart Pointers and Dynamic Arrays

The library provides a version of unique_ptr that can manage arrays allocated by new. To use a unique_ptr to manage a dynamic array, we must include a pair of empty brackets after the object type:

Click here to view code image

```
// up points to an array of ten uninitialized ints
unique_ptr<int[]> up(new int[10]);
up.release();    // automatically uses delete[] to destroy its pointer
```

The brackets in the type specifier (<int[]>) say that up points not to an int but to an array of ints. Because up points to an array, when up destroys the pointer it manages, it will automatically use delete[].

unqiue_ptrs that point to arrays provide slightly different operations than those we used in § 12.1.5 (p. 470). These operations are described in Table 12.6 (overleaf). When a unique_ptr points to an array, we cannot use the dot and arrow member access operators. After all, the unqiue_ptr points to an array, not an object so these operators would be meaningless. On the other hand, when a unqiue_ptr points to an array, we can use the subscript operator to access the elements in the array:

Click here to view code image

```
for (size_t i = 0; i != 10; ++i)
```

```
up[i] = i; // assign a new value to each of the elements
```

Table 12.6. unique_ptrs to Arrays

Member access operators (dot and arrow) are not supported for unique_ptrs to arrays. Other unique_ptr operations unchanged.	
unique_ptr<T[]> u	u can point to a dynamically allocated array of type T.
unique_ptr<T[]> u(p)	u points to the dynamically allocated array to which the built-in pointer p points. p must be convertible to T* (§ 4.11.2, p. 161).
u[i]	Returns the object at position i in the array that u owns. u must point to an array.

Unlike `unique_ptr`, `shared_ptrs` provide no direct support for managing a dynamic array. If we want to use a `shared_ptr` to manage a dynamic array, we must provide our own deleter:

Click here to view code image

```
// to use a shared_ptr we must supply a deleter
shared_ptr<int> sp(new int[10], [](int *p) { delete[] p; });
sp.reset(); // uses the lambda we supplied that uses delete[] to free the array
```

Here we pass a lambda (§ 10.3.2, p. 388) that uses `delete[]` as the deleter.

Had we neglected to supply a deleter, this code would be undefined. By default, `shared_ptr` uses `delete` to destroy the object to which it points. If that object is a dynamic array, using `delete` has the same kinds of problems that arise if we forget to use `[]` when we delete a pointer to a dynamic array (§ 12.2.1, p. 479).

The fact that `shared_ptr` does not directly support managing arrays affects how we access the elements in the array:

Click here to view code image

```
// shared_ptrs don't have subscript operator and don't support pointer arithmetic
for (size_t i = 0; i != 10; ++i)
    *(sp.get() + i) = i;  // use get to get a built-in pointer
```

There is no subscript operator for `shared_ptrs`, and the smart pointer types do not support pointer arithmetic. As a result, to access the elements in the array, we must use `get` to obtain a built-in pointer, which we can then use in normal ways.

Exercises Section 12.2.1

Exercise 12.23: Write a program to concatenate two string literals, putting the result in a dynamically allocated array of `char`. Write a program to concatenate two library `strings` that have the same value as the literals used in the first program.

Exercise 12.24: Write a program that reads a string from the standard input

into a dynamically allocated character array. Describe how your program handles varying size inputs. Test your program by giving it a string of data that is longer than the array size you've allocated.

Exercise 12.25: Given the following `new` expression, how would you `delete pa`?

```
int *pa = new int[10];
```

12.2.2. The `allocator` Class

An aspect of `new` that limits its flexibility is that `new` combines allocating memory with constructing object(s) in that memory. Similarly, `delete` combines destruction with deallocation. Combining initialization with allocation is usually what we want when we allocate a single object. In that case, we almost certainly know the value the object should have.

When we allocate a block of memory, we often plan to construct objects in that memory as needed. In this case, we'd like to decouple memory allocation from object construction. Decoupling construction from allocation means that we can allocate memory in large chunks and pay the overhead of constructing the objects only when we actually need to create them.

In general, coupling allocation and construction can be wasteful. For example:

Click here to view code image

```
string *const p = new string[n]; // construct n empty strings
string s;
string *q = p;                   // q points to the first string
while (cin >> s && q != p + n)
    *q++ = s;                    // assign a new value to *q
const size_t size = q - p;       // remember how many strings we read
// use the array
delete[] p;  // p points to an array; must remember to use delete[]
```

This `new` expression allocates and initializes `n strings`. However, we might not need `n strings`; a smaller number might suffice. As a result, we may have created objects that are never used. Moreover, for those objects we do use, we immediately assign new values over the previously initialized `strings`. The elements that are used are written twice: first when the elements are default initialized, and subsequently when we assign to them.

More importantly, classes that do not have default constructors cannot be dynamically allocated as an array.

The allocator Class

The library **allocator** class, which is defined in the `memory` header, lets us separate allocation from construction. It provides type-aware allocation of raw, unconstructed, memory. Table 12.7 (overleaf) outlines the operations that `allocator` supports. In this section, we'll describe the `allocator` operations. In § 13.5 (p. 524), we'll see an example of how this class is typically used.

Table 12.7. Standard allocator Class and Customized Algorithms

`allocator<T> a`	Defines an `allocator` object named `a` that can allocate memory for objects of type `T`.
`a.allocate(n)`	Allocates raw, unconstructed memory to hold `n` objects of type `T`.
`a.deallocate(p, n)`	Deallocates memory that held `n` objects of type `T` starting at the address in the `T*` pointer `p`; `p` must be a pointer previously returned by `allocate`, and `n` must be the size requested when `p` was created. The user must run `destroy` on any objects that were constructed in this memory before calling `deallocate`.
`a.construct(p, args)`	`p` must be a pointer to type `T` that points to raw memory; *args* are passed to a constructor for type `T`, which is used to construct an object in the memory pointed to by `p`.
`a.destroy(p)`	Runs the destructor (§ 12.1.1, p. 452) on the object pointed to by the `T*` pointer `p`.

Like `vector`, `allocator` is a template (§ 3.3, p. 96). To define an `allocator` we must specify the type of objects that a particular `allocator` can allocate. When an `allocator` object allocates memory, it allocates memory that is appropriately sized and aligned to hold objects of the given type:

Click here to view code image

```
allocator<string> alloc;            // object that can allocate strings
auto const p = alloc.allocate(n); // allocate n unconstructed strings
```

This call to `allocate` allocates memory for `n strings`.

allocators Allocate Unconstructed Memory

The memory an `allocator` allocates is *unconstructed*. We use this memory by constructing objects in that memory. In the new library the `construct` member takes a pointer and zero or more additional arguments; it constructs an element at the given location. The additional arguments are used to initialize the object being constructed. Like the arguments to `make_shared` (§ 12.1.1, p. 451), these additional arguments must be valid initializers for an object of the type being constructed. In particular, if the, object is a class type, these arguments must match a constructor for that class:

C++ 11

Click here to view code image

```
auto q = p; // q will point to one past the last constructed element
alloc.construct(q++);              // *q is the empty string
alloc.construct(q++, 10, 'c');     // *q is cccccccccc
alloc.construct(q++, "hi");        // *q is hi!
```

In earlier versions of the library, `construct` took only two arguments: the pointer at which to construct an object and a value of the element type. As a result, we could only copy an element into unconstructed space, we could not use any other constructor for the element type.

It is an error to use raw memory in which an object has not been constructed:

Click here to view code image

```
cout << *p << endl; // ok: uses the string output operator
cout << *q << endl; // disaster: q points to unconstructed memory!
```

> **Warning**
>
> We must `construct` objects in order to use memory returned by `allocate`. Using unconstructed memory in other ways is undefined.

When we're finished using the objects, we must destroy the elements we constructed, which we do by calling `destroy` on each constructed element. The `destroy` function takes a pointer and runs the destructor (§ 12.1.1, p. 452) on the pointed-to object:

Click here to view code image

```
while (q != p)
     alloc.destroy(--q);           // free the strings we actually allocated
```

At the beginning of our loop, `q` points one past the last constructed element. We decrement `q` before calling `destroy`. Thus, on the first call to `destroy`, `q` points to the last constructed element. We `destroy` the first element in the last iteration, after which `q` will equal `p` and the loop ends.

> **Warning**
>
> We may `destroy` only elements that are actually constructed.

Once the elements have been destroyed, we can either reuse the memory to hold other `strings` or return the memory to the system. We free the memory by calling `deallocate`:

```
alloc.deallocate(p, n);
```

The pointer we pass to `deallocate` cannot be null; it must point to memory allocated by `allocate`. Moreover, the size argument passed to `deallocate` must be the same size as used in the call to `allocate` that obtained the memory to which the pointer points.

Algorithms to Copy and Fill Uninitialized Memory

As a companion to the `allocator` class, the library also defines two algorithms that can construct objects in uninitialized memory. These functions, described in Table 12.8, are defined in the `memory` header.

Table 12.8. allocator Algorithms

These functions construct elements in the destination, rather than assigning to them.
`uninitialized_copy(b, e, b2)` Copies elements from the input range denoted by iterators `b` and `e` into unconstructed, raw memory denoted by the iterator `b2`. The memory denoted by `b2` must be large enough to hold a copy of the elements in the input range.
`uninitialized_copy_n(b, n, b2)` Copies `n` elements starting from the one denoted by the iterator `b` into raw memory starting at `b2`.
`uninitialized_fill(b, e, t)` Constructs objects in the range of raw memory denoted by iterators `b` and `e` as a copy of `t`.
`uninitialized_fill_n(b, n, t)` Constructs an `unsigned` number `n` objects starting at `b`. `b` must denote unconstructed, raw memory large enough to hold the given number of objects.

As an example, assume we have a `vector` of `ints` that we want to copy into dynamic memory. We'll allocate memory for twice as many `ints` as are in the `vector`. We'll construct the first half of the newly allocated memory by copying elements from the original `vector`. We'll construct elements in the second half by filling them with a given value:

Click here to view code image

```
// allocate twice as many elements as vi holds
auto p = alloc.allocate(vi.size() * 2);
// construct elements starting at p as copies of elements in vi
auto q = uninitialized_copy(vi.begin(), vi.end(), p);
// initialize the remaining elements to 42
uninitialized_fill_n(q, vi.size(), 42);
```

Like the `copy` algorithm (§ 10.2.2, p. 382), `uninitialized_copy` takes three iterators. The first two denote an input sequence and the third denotes the destination into which those elements will be copied. The destination iterator passed to `uninitialized_copy` must denote unconstructed memory. Unlike `copy`, `uninitialized_copy` constructs elements in its destination.

Like `copy`, `uninitialized_copy` returns its (incremented) destination iterator. Thus, a call to `uninitialized_copy` returns a pointer positioned one element past the last constructed element. In this example, we store that pointer in `q`, which we pass to `uninitialized_fill_n`. This function, like `fill_n` (§ 10.2.2, p. 380), takes a pointer to a destination, a count, and a value. It will construct the given number of objects from the given value at locations starting at the given destination.

Exercises Section 12.2.2

Exercise 12.26: Rewrite the program on page 481 using an `allocator`.

12.3. Using the Library: A Text-Query Program

To conclude our discussion of the library, we'll implement a simple text-query program. Our program will let a user search a given file for words that might occur in it. The result of a query will be the number of times the word occurs and a list of lines on which that word appears. If a word occurs more than once on the same line, we'll display that line only once. Lines will be displayed in ascending order—that is, line 7 should be displayed before line 9, and so on.

For example, we might read the file that contains the input for this chapter and look for the word `element`. The first few lines of the output would be

Click here to view code image

```
element occurs 112 times
    (line 36) A set element contains only a key;
    (line 158) operator creates a new element
    (line 160) Regardless of whether the element
    (line 168) When we fetch an element from a map, we
    (line 214) If the element is not found, find returns
```

followed by the remaining 100 or so lines in which the word `element` occurs.

12.3.1. Design of the Query Program

A good way to start the design of a program is to list the program's operations. Knowing what operations we need can help us see what data structures we'll need. Starting from requirements, the tasks our program must do include the following:

- When it reads the input, the program must remember the line(s) in which each word appears. Hence, the program will need to read the input a line at a time and break up the lines from the input file into its separate words
- When it generates output,
  - The program must be able to fetch the line numbers associated with a given word
  - The line numbers must appear in ascending order with no duplicates
  - The program must be able to print the text appearing in the input file at a given line number.

These requirements can be met quite neatly by using various library facilities:

- We'll use a `vector<string>` to store a copy of the entire input file. Each line in the input file will be an element in this `vector`. When we want to print a line, we can fetch the line using its line number as the index.
- We'll use an `istringstream` (§ 8.3, p. 321) to break each line into words.
- We'll use a `set` to hold the line numbers on which each word in the input appears. Using a `set` guarantees that each line will appear only once and that the line numbers will be stored in ascending order.
- We'll use a `map` to associate each word with the `set` of line numbers on which the word appears. Using a `map` will let us fetch the `set` for any given word.

For reasons we'll explain shortly, our solution will also use `shared_ptrs`.

Data Structures

Although we could write our program using `vector, set`, and `map` directly, it will be more useful if we define a more abstract solution. We'll start by designing a class to hold the input file in a way that makes querying the file easy. This class, which we'll name `TextQuery`, will hold a `vector` and a `map`. The `vector` will hold the text of the input file; the `map` will associate each word in that file to the `set` of line numbers on which that word appears. This class will have a constructor that reads a given input file and an operation to perform the queries.

The work of the query operation is pretty simple: It will look inside its `map` to see whether the given word is present. The hard part in designing this function is deciding what the query function should return. Once we know that a word was found, we need to know how often it occurred, the line numbers on which it occurred, and the corresponding text for each of those line numbers.

The easiest way to return all those data is to define a second class, which we'll name `QueryResult`, to hold the results of a query. This class will have a `print`

function to print the results in a `QueryResult`.

Sharing Data between Classes

Our `QueryResult` class is intended to represent the results of a query. Those results include the `set` of line numbers associated with the given word and the corresponding lines of text from the input file. These data are stored in objects of type `TextQuery`.

Because the data that a `QueryResult` needs are stored in a `TextQuery` object, we have to decide how to access them. We could copy the `set` of line numbers, but that might be an expensive operation. Moreover, we certainly wouldn't want to copy the `vector`, because that would entail copying the entire file in order to print (what will usually be) a small subset of the file.

We could avoid making copies by returning iterators (or pointers) into the `TextQuery` object. However, this approach opens up a pitfall: What happens if the `TextQuery` object is destroyed before a corresponding `QueryResult`? In that case, the `QueryResult` would refer to data in an object that no longer exists.

This last observation about synchronizing the lifetime of a `QueryResult` with the `TextQuery` object whose results it represents suggests a solution to our design problem. Given that these two classes conceptually "share" data, we'll use `shared_ptrs` (§ 12.1.1, p. 450) to reflect that sharing in our data structures.

Using the TextQuery Class

When we design a class, it can be helpful to write programs using the class before actually implementing the members. That way, we can see whether the class has the operations we need. For example, the following program uses our proposed `TextQuery` and `QueryResult` classes. This function takes an `ifstream` that points to the file we want to process, and interacts with a user, printing the results for the given words:

Click here to view code image

```
void runQueries(ifstream &infile)
{
    // infile is an ifstream that is the file we want to query
    TextQuery tq(infile);  // store the file and build the query map
    // iterate with the user: prompt for a word to find and print results
    while (true) {
        cout << "enter word to look for, or q to quit: ";
        string s;
        // stop if we hit end-of-file on the input or if a 'q' is entered
        if (!(cin >> s) || s == "q") break;
        // run the query and print the results
        print(cout, tq.query(s)) << endl;
    }
```

```
}
```

We start by initializing a `TextQuery` object named `tq` from a given `ifstream`. The `TextQuery` constructor reads that file into its `vector` and builds the `map` that associates the words in the input with the line numbers on which they appear.

The `while` loop iterates (indefinitely) with the user asking for a word to query and printing the related results. The loop condition tests the literal `true` (§ 2.1.3, p. 41), so it always succeeds. We exit the loop through the `break` (§ 5.5.1, p. 190) after the first `if`. That `if` checks that the read succeeded. If so, it also checks whether the user entered a `q` to quit. Once we have a word to look for, we ask `tq` to find that word and then call `print` to print the results of the search.

Exercises Section 12.3.1

Exercise 12.27: The `TextQuery` and `QueryResult` classes use only capabilities that we have already covered. Without looking ahead, write your own versions of these classes.

Exercise 12.28: Write a program to implement text queries without defining classes to manage the data. Your program should take a file and interact with a user to query for words in that file. Use `vector`, `map`, and `set` containers to hold the data for the file and to generate the results for the queries.

Exercise 12.29: We could have written the loop to manage the interaction with the user as a `do while` (§ 5.4.4, p. 189) loop. Rewrite the loop to use a `do while`. Explain which version you prefer and why.

12.3.2. Defining the Query Program Classes

We'll start by defining our `TextQuery` class. The user will create objects of this class by supplying an `istream` from which to read the input file. This class also provides the `query` operation that will take a `string` and return a `QueryResult` representing the lines on which that `string` appears.

The data members of the class have to take into account the intended sharing with `QueryResult` objects. The `QueryResult` class will share the `vector` representing the input file and the `sets` that hold the line numbers associated with each word in the input. Hence, our class has two data members: a `shared_ptr` to a dynamically allocated `vector` that holds the input file, and a `map` from `string` to `shared_ptr<set>`. The `map` associates each word in the file with a dynamically allocated `set` that holds the line numbers on which that word appears.

To make our code a bit easier to read, we'll also define a type member (§ 7.3.1, p. 271) to refer to line numbers, which are indices into a `vector` of `strings`:

Click here to view code image

```
class QueryResult; // declaration needed for return type in the query function
class TextQuery {
public:
    using line_no = std::vector<std::string>::size_type;
    TextQuery(std::ifstream&);
    QueryResult query(const std::string&) const;
private:
     std::shared_ptr<std::vector<std::string>> file;   // input
file
    // map of each word to the set of the lines in which that word appears
    std::map<std::string,
             std::shared_ptr<std::set<line_no>>> wm;
};
```

The hardest part about this class is untangling the class names. As usual, for code that will go in a header file, we use `std::` when we use a library name (§ 3.1, p. 83). In this case, the repeated use of `std::` makes the code a bit hard to read at first. For example,

Click here to view code image

```
std::map<std::string, std::shared_ptr<std::set<line_no>>> wm;
```

is easier to understand when rewritten as

Click here to view code image

```
map<string, shared_ptr<set<line_no>>> wm;
```

The TextQuery Constructor

The `TextQuery` constructor takes an `ifstream`, which it reads a line at a time:

Click here to view code image

```
// read the input file and build the map of lines to line numbers
TextQuery::TextQuery(ifstream &is): file(new vector<string>)
{
    string text;
    while (getline(is, text)) {        // for each line in the file
        file->push_back(text);         // remember this line of text
        int n = file->size() - 1;      // the current line number
        istringstream line(text);      // separate the line into words
        string word;
        while (line >> word) {         // for each word in that line
            // if word isn't already in wm, subscripting adds a new entry
            auto &lines = wm[word]; // lines is a shared_ptr
            if (!lines) // that pointer is null the first time we see word
```

```
                lines.reset(new set<line_no>); // allocate a new
set
            lines->insert(n);       // insert this line number
        }
    }
}
```

The constructor initializer allocates a new `vector` to hold the text from the input file. We use `getline` to read the file a line at a time and push each line onto the `vector`. Because `file` is a `shared_ptr`, we use the `->` operator to dereference `file` to fetch the `push_back` member of the `vector` to which `file` points.

Next we use an `istringstream` (§ 8.3, p. 321) to process each word in the line we just read. The inner `while` uses the `istringstream` input operator to read each word from the current line into `word`. Inside the `while`, we use the `map` subscript operator to fetch the `shared_ptr<set>` associated with `word` and bind `lines` to that pointer. Note that `lines` is a reference, so changes made to `lines` will be made to the element in `wm`.

If `word` wasn't in the `map`, the subscript operator adds `word` to `wm` (§ 11.3.4, p. 435). The element associated with `word` is value initialized, which means that `lines` will be a null pointer if the subscript operator added `word` to `wm`. If `lines` is null, we allocate a `new set` and call `reset` to update the `shared_ptr` to which `lines` refers to point to this newly allocated `set`.

Regardless of whether we created a new `set`, we call `insert` to add the current line number. Because `lines` is a reference, the call to `insert` adds an element to the `set` in `wm`. If a given word occurs more than once in the same line, the call to `insert` does nothing.

The QueryResult Class

The `QueryResult` class has three data members: a `string` that is the word whose results it represents; a `shared_ptr` to the `vector` containing the input file; and a `shared_ptr` to the `set` of line numbers on which this word appears. Its only member function is a constructor that initializes these three members:

Click here to view code image

```
class QueryResult {
friend      std::ostream&      print(std::ostream&,      const
QueryResult&);
public:
    QueryResult(std::string s,
                std::shared_ptr<std::set<line_no>> p,
                std::shared_ptr<std::vector<std::string>> f):
        sought(s), lines(p), file(f) { }
private:
    std::string sought;  // word this query represents
```

```
    std::shared_ptr<std::set<line_no>> lines; // lines it's on
    std::shared_ptr<std::vector<std::string>> file; // input file
};
```

The constructor's only job is to store its arguments in the corresponding data members, which it does in the constructor initializer list (§ 7.1.4, p. 265).

The query Function

The `query` function takes a `string`, which it uses to locate the corresponding `set` of line numbers in the `map`. If the `string` is found, the `query` function constructs a `QueryResult` from the given `string`, the `TextQuery file` member, and the `set` that was fetched from `wm`.

The only question is: What should we return if the given `string` is not found? In this case, there is no `set` to return. We'll solve this problem by defining a local `static` object that is a `shared_ptr` to an empty `set` of line numbers. When the word is not found, we'll return a copy of this `shared_ptr`:

Click here to view code image

```
QueryResult
TextQuery::query(const string &sought) const
{
    // we'll return a pointer to this set if we don't find sought
    static shared_ptr<set<line_no>> nodata(new set<line_no>);
    // use find and not a subscript to avoid adding words to wm!
    auto loc = wm.find(sought);
    if (loc == wm.end())
        return QueryResult(sought, nodata, file); // not found
    else
        return QueryResult(sought, loc->second, file);
}
```

Printing the Results

The `print` function prints its given `QueryResult` object on its given stream:

Click here to view code image

```
ostream &print(ostream & os, const QueryResult &qr)
{
    // if the word was found, print the count and all occurrences
    os << qr.sought << " occurs " << qr.lines->size() << " "
       << make_plural(qr.lines->size(), "time", "s") <<
endl;
    // print each line in which the word appeared
    for (auto num : *qr.lines) // for every element in the set
        // don't confound the user with text lines starting at 0
```

```
            os << "\t(line " << num + 1 << ") "
               << *(qr.file->begin() + num) << endl;
        return os;
    }
```

We use the `size` of the `set` to which the `qr.lines` points to report how many matches were found. Because that `set` is in a `shared_ptr`, we have to remember to dereference `lines`. We call `make_plural` (§ 6.3.2, p. 224) to print `time` or `times`, depending on whether that size is equal to 1.

In the `for` we iterate through the `set` to which `lines` points. The body of the `for` prints the line number, adjusted to use human-friendly counting. The numbers in the `set` are indices of elements in the `vector`, which are numbered from zero. However, most users think of the first line as line number 1, so we systematically add 1 to the line numbers to convert to this more common notation.

We use the line number to fetch a line from the `vector` to which `file` points. Recall that when we add a number to an iterator, we get the element that many elements further into the `vector` (§ 3.4.2, p. 111). Thus, `file->begin() + num` is the `num`th element after the start of the `vector` to which `file` points.

Note that this function correctly handles the case that the word is not found. In this case, the `set` will be empty. The first output statement will note that the word occurred 0 times. Because `*res.lines` is empty. the `for` loop won't be executed.

Exercises Section 12.3.2

Exercise 12.30: Define your own versions of the `TextQuery` and `QueryResult` classes and execute the `runQueries` function from § 12.3.1 (p. 486).

Exercise 12.31: What difference(s) would it make if we used a `vector` instead of a `set` to hold the line numbers? Which approach is better? Why?

Exercise 12.32: Rewrite the `TextQuery` and `QueryResult` classes to use a `StrBlob` instead of a `vector<string>` to hold the input file.

Exercise 12.33: In Chapter 15 we'll extend our query system and will need some additional members in the `QueryResult` class. Add members named `begin` and `end` that return iterators into the `set` of line numbers returned by a given query, and a member named `get_file` that returns a `shared_ptr` to the file in the `QueryResult` object.

Chapter Summary

In C++, memory is allocated through `new` expressions and freed through `delete` expressions. The library also defines an `allocator` class for allocating blocks of dynamic memory.

Programs that allocate dynamic memory are responsible for freeing the memory they allocate. Properly freeing dynamic memory is a rich source of bugs: Either the memory is never freed, or it is freed while there are still pointers referring to the memory. The new library defines smart pointers—`shared_ptr`, `unique_ptr`, and `weak_ptr`—that make managing dynamic memory much safer. A smart pointer automatically frees the memory once there are no other users of that memory. When possible, modern C++ programs ought to use smart pointers.

Defined Terms

allocator Library class that allocates unconstructed memory.

dangling pointer A pointer that refers to memory that once had an object but no longer does. Program errors due to dangling pointers are notoriously difficult to debug.

delete Frees memory allocated by `new`. `delete p` frees the object and `delete [] p` frees the array to which `p` points. `p` may be null or point to memory allocated by `new`.

deleter Function passed to a smart pointer to use in place of `delete` when destroying the object to which the pointer is bound.

destructor Special member function that cleans up an object when the object goes out of scope or is deleted.

dynamically allocated Object that is allocated on the free store. Objects allocated on the free store exist until they are explicitly deleted or the program terminates.

free store Memory pool available to a program to hold dynamically allocated objects.

heap Synonym for free store.

new Allocates memory from the free store. `new T` allocates and constructs an object of type `T` and returns a pointer to that object; if `T` is an array type, `new` returns a pointer to the first element in the array. Similarly, `new [n] T` allocates *n* objects of type `T` and returns a pointer to the first element in the array. By default, the allocated object is default initialized. We may also provide optional initializers.

placement new Form of `new` that takes additional arguments passed in parentheses following the keyword `new`; for example, `new (nothrow) int` tells `new` that it should not throw an exception.

reference count Counter that tracks how many users share a common object.

Used by smart pointers to know when it is safe to delete memory to which the pointers point.

shared_ptr Smart pointer that provides shared ownership: The object is deleted when the last `shared_ptr` pointing to that object is destroyed.

smart pointer Library type that acts like a pointer but can be checked to see whether it is safe to use. The type takes care of deleting memory when appropriate.

unique_ptr Smart pointer that provides single ownership: The object is deleted when the `unique_ptr` pointing to that object is destroyed. `unique_ptrs` cannot be directly copied or assigned.

weak_ptr Smart pointer that points to an object managed by a `shared_ptr`. The `shared_ptr` does not count `weak_ptrs` when deciding whether to delete its object.

Part III: Tools for Class Authors

Contents

Classes are the central concept in C++. Chapter 7 began our detailed coverage of how classes are defined. That chapter covered topics fundamental to any use of classes: class scope, data hiding, and constructors. It also introduced various important class features: member functions, the implicit `this` pointer, friends, and `const, static`, and `mutable` members. In this part, we'll extend our coverage of classes by looking at copy control, overloaded operators, inheritance, and templates.

As we've seen, in C++ classes define constructors to control what happens when objects of the class type are initialized. Classes also control what happens when objects are copied, assigned, moved, and destroyed. In this respect, C++ differs from other languages, many of which do not give class designers the ability to control these operations. Chapter 13 covers these topics. This chapter also covers two important concepts introduced by the new standard: rvalue references and move operations.

Chapter 14 looks at operator overloading, which allows operands of class types to be used with the built-in operators. Operator overloading is one of the ways whereby C++ lets us create new types that are as intuitive to use as are the built-in types.

Among the operators that a class can overload is the funtion call operator. We can