Chapter 1

Exercises Section 1.2.2

Exercise 1.5: *We wrote the output in one large statement. Rewrite the program to use a separate statement to print each operand.*

Answer 1.5:

```
std::cout << "The sum of ";
std::cout << v1;
std::cout << " and ";
std::cout << v2;
std::cout << " is ";
std::cout << v1 + v2;
std::cout << std::endl;
```

Exercise 1.6: *Explain what the following program fragment does:*

```
std::cout << "The sum of " << v1;
          << " and " << v2;
          << " is " << v1 + v2
          << std::endl;
```

Is this code legal? If so, why? If not, why not?

Answer 1.6: The code is not legal. The semicolon after `v1` ends the output statement. That makes the next line illegal. Assuming we fixed that problem, the semicolon after `v2` also ends a statement, making the third line illegal. To correct the code, we could either remove both semicolons or repeat `std::cout` on the following lines:

```
// first possible fix: remove the spurious semicolons
std::cout << "The sum of " << v1    // semicolon removed
          << " and " << v2          // semicolon removed
          << " is " << v1 + v2
          << std::endl;

// second fix: make the first three lines separate statements
std::cout << "The sum of " << v1;
std::cout << " and " << v2;          // std::cout added
std::cout << " is " << v1 + v2       // std::cout added
          << std::endl;
```

Once corrected, the program prints the sum of `v1` and `v2` along with text indicating that fact. Assuming `v1` is `3` and `v2` is `7`, the program would print:

```
The sum of 3 and 7 is 10
```

Exercises Section 1.3

Exercise 1.8: *Indicate which, if any, of the following output statements, are legal.*

```
std::cout << "/*";
std::cout << "*/";
std::cout << /* "*/" */;
```

After you've predicted what will happen, test your answer by compiling a program with these three statements. Correct any errors you encounter.

Answer 1.8: The first two statements are legal. The third is illegal. Although the first `*/` appears inside a string literal, the compiler treats the `*/` as a close comment delimiter. Essentially, this statement is treated as if we'd written:

```
std::cout << " */;
```

which is an attempt to print an incomplete string literal.

Exercises Section 1.4.2

Exercise 1.9: *What does the following* `for` *loop do? What is the final value of* `sum`*?*

```
int sum = 0;
for (int i = -100; i <= 100; ++i)
    sum += i;
```

Answer 1.9: The loop adds the integers from `-100` through `100`. The result is 0.

Exercises Section 1.4.3

Exercise 1.14: *What happens in the program presented in this section if the input values are equal?*

Answer 1.14: If the input values are equal, then the loop is executed once, which has the effect of adding that value to 0.

Exercise 1.17: *Write a program to ask the user to enter a series of numbers. Print a message saying how many of the numbers are negative numbers.*

Answer 1.17:

```
#include <iostream>
int main()
{
    std::cout << "Enter a series of numbers" << std::endl;
    int i;
    int ctr = 0;
    while (std::cin >> i)
        if (i < 0)
            ++ctr;
    std::cout << ctr << " numbers were negative" << std::endl;
    return 0;
}
```

Exercises Section 1.4.4

Exercise 1.19: *What happens if you give the numbers 1000 and 2000 to the program written for the previous exercise? Revise the program so that it never prints more than 10 numbers per line.*

Answer 1.19: What happens when the input values are 1000 and 2000 depends on the details of the program's print statement. The program might well have printed all its output to a single line in which case the output line would be enormous—it would print the thousand values from 1000 up to 2000. If it printed a single number to each line, then the output would consume 1000 lines.

We could force the program to print 10 values to a line as follows:

```
#include <iostream>
int main()
{
```

```
    std::cout << "Enter two numbers:" << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2; // read input
    // use smaller number as lower bound for printing the range
    // and larger number as upper bound
    int lower, upper;
    if (v1 <= v2) {
        lower = v1;
        upper = v2;
    } else {
        lower = v2;
        upper = v1;
    }
    // print values from lower up to and but not including upper
    int newline_ctr = 0;
    for (int val = lower; val != upper; ++val) {
        std::cout << val << " ";
        ++newline_ctr;            // increment counter to indicate another value printed
        if (newline_ctr == 10) {  // if we've already printed 10 values
            std::cout << std::endl;   // print this line
            newline_ctr = 0;          // and reset the counter to 0
        }
    }

    std::cout << std::endl;
    return 0;
}
```

Exercises Section 1.5.1

Exercise 1.23: *Write a program that reads several transactions for the same* ISBN*. Write the sum of all the transactions that were read.*

Answer 1.23:

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item result, next_item;
    std::cin >> result;
    while (std::cin >> next_item)       // read the next transaction
        result = result + next_item;
    std::cout << result << std::endl;   // print the total
    return 0;
}
```

Exercises Section 1.6

Exercise 1.26: *In the bookstore program we used the addition operator and not the compound assignment operator to add* `trans` *to* `total`*. Why didn't we use the compound assignment operator?*

Answer 1.26: The only operations that we can use on objects of type `Sales_item` are those listed in section 1.5.1 (p. 21). Those operations include the addition operator but do not include the compound assignment operator.

Chapter 2

Exercises Section 2.1.2

Exercise 2.3: *If a* `short` *on a given machine has 16 bits then what is the largest number that can be assigned to a* `short`*? To an* `unsigned short`*?*

Answer 2.3: 32767 can be assigned to a `short`, 65535 to an `unsigned short`.

Exercise 2.4: *What value is assigned if we assign 100,000 to a 16-bit* `unsigned short`*? What value is assigned if we assign 100,000 to a plain 16-bit* `short`*?*

Answer 2.4: 100000 is larger than the largest value that can be stored in a 16-bit `unsigned short`. The largest value that will fit is 65535, so the value assigned will be 34464, which is 100000 modulo 65536.

There are no guarantees about what value will be used when we assign 100000 to a plain (`signed`) `short`.

Exercise 2.6: *To calculate a mortgage payment, what types would you use for the rate, principal, and payment? Explain why you selected each type.*

Answer 2.6: For each of these we would use a `double`. The `float` type is allowed to be as small as 6 significant digits, which is too small for the principal. A `float` arguably would be sufficient for holding the interest rate, which ordinarily has at most 4 significant digits and is likely to be sufficient for the payment, so long as the payment is less than `$10,000`. However, there is no advantage to using `float`, so we might just as well use `double` for all three values.

Exercises Section 2.2

Exercise 2.8: *Determine the type of each of these literal constants:*

```
(a) -10   (b) -10u   (c) -10.   (d) -10e-2
```

Answer 2.8: (a) `int` (b) `unsigned int` (c) `double` (d) `double`

Exercises Section 2.3.1

Exercise 2.12: *Distinguish between an lvalue and an rvalue; show examples of each.*

Answer 2.12: An lvalue is an expression that represents a region of memory and which may appear on the left-hand or right-hand side of an assignment. An rvalue may only appear as the right-hand operand of assignment: We can write to an lvalue; an rvalue is a value that we can read, but may not write.

A variable is an lvalue; a numeric literal is an rvalue.

Exercise 2.13: *Name one case where an lvalue is required.*

Answer 2.13: The left-hand operand of an assignment must be an lvalue.

Exercises Section 2.3.2

Exercise 2.14: *Which, if any, of the following names are invalid? Correct each identified invalid name.*

```
(a) int double = 3.14159;       (b) char _;
(c) bool catch-22;              (d) char 1_or_2 = '1';
(e) float Float = 3.14f;
```

Answer 2.14:

(a) `double` is a reserved word; it may not be an identifier. We might change the name in a variety of ways: `Double` or `dbl` are two obvious ways to do so. Depending on the naming convention, `Double` might be misleading. Uppercase identifiers are often used to denote types rather than variables.

(b) _ is a legal, albeit confusing, identifier.

(c) `catch-22` is invalid; names may not include hyphens. The easiest way to correct this name would be to use an underscore in place of the hyphen, `catch_22`.

(d) `1_or_2` is invalid; an identifier may not begin with a digit. We might correct this name by spelling the digits: `one_or_two`.

(e) `Float` is a legal identifier.

Exercises Section 2.3.3

Exercise 2.15: *What, if any, are the differences between the following definitions:*

```
int month = 9, day = 7;

int month = 09, day = 07;
```

If either definition contains an error, how might you correct the problem?

Answer 2.15: The second definition of `month` is illegal. The literal values used as the initializers begin with a `0`, which means that they are octal literals. There is no octal digit `9`, so the initializer is invalid.

The initialization of `day` is the same in both cases, because octal `07` and decimal `7` are the same value.

Exercises Section 2.3.4

Exercise 2.17: *What are the initial values, if any, of each of the following variables?*

```
std::string global_str;
int global_int;
int main()
{
    int local_int;
    std::string local_str;
    // ...
    return 0;
}
```

Answer 2.17: `global_str` and `local_str` are each an empty string. `global_int` has value 0; the value of `local_int` is undefined.

Exercises Section 2.3.6

Exercise 2.19: *What is the value of* `j` *in the following program?*

```
int i = 42;
int main()
{
    int i = 100;
    int j = i;
    // ...
}
```

Answer 2.19: The value of `j` is 100. The outer variable named `i` is hidden by the local variable with the same name.

Exercise 2.21: *Is the following program legal?*

```
int sum = 0;
for (int i = 0; i != 10; ++i)
    sum += i;
std::cout << "Sum from 0 to " << i
          << " is " << sum << std::endl;
```

Answer 2.21: No, the program is illegal, although some compilers may not detect the error. The problem is that `i` is local to the `for` statement, so `i` is inaccessible to the output statement. Prior to standard C++ this program would be legal. Compilers that do not yet match the standard might accept this code.

Exercises Section 2.4

Exercise 2.23: *Which of the following are legal? For those usages that are illegal, explain why.*

```
(a) const int buf;
(b) int cnt = 0;
    const int sz = cnt;
(c) cnt++; sz++;
```

Answer 2.23:
(a) illegal—`const` variables must be initialized.
(b) legal
(c) illegal—`sz` is `const`; it may not be incremented.

Exercises Section 2.5

Exercise 2.27: *What does the following code print?*

```
int i, &ri = i;
i = 5; ri = 10;
std::cout << i << " " << ri << std::endl;
```

Answer 2.27: The program prints:

```
10 10
```

The assignment to `ri` is an assignment to `i` through the reference `ri`. Thus, the assignment to `ri` changes the value of `i` to 10. The output statement prints `i` twice, first directly and then indirectly through the reference.

Exercises Section 2.8

Exercise 2.30: *Define the data members of classes to represent the following types:*

(a) a phone number *(b) an address*
(c) an employee or a company *(d) a student at a university*

Answer 2.30: Note: These answers are appropriate to customs in the US. Other countries would use different members for phone numbers and addresses.

(a) `Phone` number data members:

```
int area_code;
int exchange;
int extension;
```

(b) `Address` data members:

```
std::string name;
std::string address1, address2; // street, apartment etc.
std::string city;
std::string state;
long zip;
```

(c) `Employee` data members:

```
std::string name;
Phone home_phone, office_phone;
Address home_addr, office_addr;
```

(d) `Student` data members:

```
std::string name;
Phone phone;
Address address;
unsigned int year_enrolled, year_graduated;
```

Exercises Section 2.9.1

Exercise 2.31: *Identify which of the following statements are declarations and which ones are definitions. Explain why they are declarations or definitions.*

```
(a) extern int ix = 1024;
(b) int iy;
(c) extern int iz;
(d) extern const int &ri;
```

Answer 2.31:
(a) Definition because the variable has an initializer.
(b) Definition because there is no `extern` keyword.
(c) Declaration because `extern` is used and there is no initializer.
(d) Declaration because when `extern` appears, and there is no initializer, the variable is a declaration regardless of whether it is `const`.

Exercise 2.32: *Which of the following declarations and definitions would you put in a header? In a source file? Explain why.*

```
(a) int var;
(b) const double pi = 3.1416;
(c) extern int total = 255;
(d) const double sq2 = sqrt(2.0);
```

Answer 2.32:

(a) This statement defines `var` and so most probably belongs in a source file, not a header.

(b) `const` variables that are initialized to a constant value ordinarily should go in header files. When we put definitions such as this one in a header, each file that includes the header will have its own copy of `pi`. By putting the definition in a header we give the compiler access to the value of the initializer in each file.

(c) Despite the `extern` keyword, this statement is a definition because the variable is initialized. Definitions ordinarily go in source files, not headers.

(d) Because the initializer is not itself a constant, there is no reason to put the initialization in a header file. However, we might still want to make the initialized `const` value available across multiple files. The way to do so would be to put an `extern` declaration for `sq2` into a header and put the definition of `sq2` in one of the source files that includes that header:

```
extern const double sq2;   // this declaration goes in a header

extern const double sq2 = sqrt(2.0);   // the definition goes in a source file
```

Chapter 3

Exercises Section 3.1

Exercise 3.1: *Rewrite the program from Section 2.3 (p. 43) that calculated the result of raising a given number to a given power to use appropriate* `using` *declarations rather than accessing library names through a* `std::` *prefix.*

Answer 3.1:

```
#include <iostream>
using std::cout; using std::endl;
int main()
{
    // a first, not very good, solution
    cout << "2 raised to the power of 10: ";
    cout << 2*2*2*2*2*2*2*2*2*2;
    cout << endl;
    return 0;
}
```

Exercises Section 3.2.1

Exercise 3.2: *What is a default constructor?*

Answer 3.2: The default constructor is the constructor that takes no initializers. For example:

```
string s;         // No initializer, uses the default constructor
string s2("hi");// Initializes s2 using "hi" as the initializer
```

Exercise 3.3: *Name the three ways to initialize a* `string`*.*

Answer 3.3: Actually we have already seen four:

1. As an empty `string`, using the default `string` constructor, which takes no arguments.
2. As a copy of another `string`.
3. From a character string literal.
4. From a count and a character.

Exercises Section 3.2.2

Exercise 3.5: *Write a program to read the standard input a line at a time. Modify your program to read a word at a time.*

Answer 3.5:

```
#include <iostream>
#include <string>
using std::cin; using std::cout; using std::endl;
using std::string;
int main()
{
    // read the standard input a line at a time.
    string s;
    while (getline(cin, s))
        // print what was read, remembering to print a newline
```

```
        cout << s << endl;
    return 0;
}
```

```
#include <iostream>
#include <string>
using std::cin; using std::cout; using std::endl;
using std::string;
int main()
{
    // read the standard input a word at a time.
    string s;
    while (cin >> s)
        // print each word that was read on a separate line
        cout << s << endl;
    return 0;
}
```

Exercises Section 3.2.4

Exercise 3.7: *Write a program to read two* `strings` *and report whether the* `strings` *are equal. If not, report which of the two is the larger. Now, change the program to report whether the* `strings` *have the same length, and if not, report which is longer.*

Answer 3.7:

```
#include <iostream>
#include <string>
using std::cin; using std::cout; using std::endl;
using std::string;
int main()
{
    // Compare two strings and say whether they're equal.
    // If unequal indicate which string is larger.
    string s1, s2;
    cin >> s1 >> s2;
    if (s1 == s2)
        cout << "Strings are equal" << endl;
    else {
        cout << "Strings are unequal: ";
        if (s1 > s2)
            cout << s1 << " is larger than " << s2 << endl;
        else
            cout << s2 << " is larger than " << s1 <<endl;
    }
    // Now repeat the program, but report whether the strings have the
    // same length and if not which is longer.
    if (s1.size() == s2.size())
        cout << s1 << " and " << s2 << " are the same length" << endl;
    else {
        cout << "Strings are unequal in length: ";
        if (s1.size() > s2.size())
            cout << s1 << " is longer than " << s2 << endl;
        else
            cout << s2 << " is longer than " << s1 << endl;
    }
    return 0;
}
```

Exercise 3.9: *What does the following program do? Is it valid? If not, why not?*

```
string s;
cout << s[0] << endl;
```

Answer 3.9: This code fragment is invalid; it attempts to write the (nonexistent) first character from an empty string.

The program defines and initializes `s` using the default `string` constructor. That constructor initializes `s` to the empty string, a string with no characters. The attempt to print `s[0]` is an attempt to print the character at position zero. There is no such character.

Exercises Section 3.3.1

Exercise 3.11: *Which, if any, of the following vector definitions are in error?*

```
(a) vector< vector<int> > ivec;
(b) vector<string> svec = ivec;
(c) vector<string> svec(10, "null");
```

Answer 3.11:

(a) OK; creates an empty `vector` each of whose elements is a `vector` of `int`.
(b) Error: attempts to use a `vector` that holds `vector<int>` to initialize a `vector` that holds `string`s.
(c) OK: creates a `vector` with 10 `string`s. Each `string` element is the four-character string `null`.

Exercises Section 3.3.2

Exercise 3.14: *Read some text into a* `vector`*, storing each word in the input as an element in the* `vector`*. Transform each word into uppercase letters. Print the transformed elements from the* `vector`*, printing eight words to a line.*

Answer 3.14:

```
#include <iostream>
#include <string>
#include <vector>
#include <cctype>
using std::cin; using std::cout; using std::endl;
using std::string; using std::vector; using std::toupper;
int main()
{
    string s;                   // hold each string from the input
    vector<string> vec;         // hold the uppercase versions of the input

    // first read and transform the input storing it in vec
    while (cin >> s) {
        for (string::size_type i = 0; i != s.size(); ++i)
            s[i] = toupper(s[i]);
        vec.push_back(s);
    }

    // now print the vec 8 words to a line
    vector<string>::size_type ctr = 0;    // track number of words printed
    for (vector<string>::size_type i = 0; i != vec.size(); ++i) {
        cout << vec[i];
        // check whether it's time to print a newline
        if (++ctr == 8) {
```

```
            cout << endl;
            ctr = 0;
        } else
            cout << " ";
    }
    return 0;
}
```

Exercise 3.16: *List three ways to define a* `vector` *and give it 10 elements, each with the value 42. Indicate whether there is a preferred way to do so and why.*

Answer 3.16:

1. Define a `vector` using a count and a value as the initializer:

   ```
   // Best way: Define the vector directly to hold 10 elements,
   //          initializing each element to the value 42
   vector<int> one(10, 42);
   ```

   This approach is the simplest and most direct way to define a `vector` with a given number of elements all of which have the same initial value. It is the best way to solve the problem we were given.

2. A second way to accomplish our task is to define an empty `vector` and then use `push_back` to add elements.

   ```
   // Ok, but not optimal when we know the initial values:
   //          Define an empty vector and then add elements with
   //          the appropriate values
   vector<int> two;
   for (int i = 0; i < 10; ++i)
       two.push_back(42);
   ```

3. The third way would be to define a `vector` holding 10 elements and then assign the value 42 to each element. This approach is least optimal: It defines and initializes the elements and then assigns a new value to each element:

   ```
   // Least good: Define the vector to hold 10 elements,
   //            initializing each element to the value 0
   vector<int> three(10);
   for (int i = 0; i < 10; ++i)
       three[i] = 42;
   ```

   This approach touches each element twice—first when the element is initialized to 0 and then when the desired value (42) is assigned to the element.

Exercises Section 3.4

Exercise 3.17: *Redo the exercises from Section 3.3.2 (p. 96), using iterators rather than subscripts to access the elements in the* `vector`*.*

Answer 3.17: As one example, here is the program for exercise 3.14 rewritten to use a `vector` iterator:

```
#include <iostream>
#include <string>
#include <vector>
#include <cctype>
using std::cin; using std::cout; using std::endl;
using std::string; using std::vector; using std::toupper;
```

```
int main()
{
    // no need to change the first part of the program,
    // which reads and populates the vector
    string s;                // hold each string from the input
    vector<string> vec;      // hold the uppercase versions of the input

    // first read and transform the input storing it in vec
    while (cin >> s) {
        for (string::size_type i = 0; i != s.size(); ++i)
            s[i] = toupper(s[i]);
        vec.push_back(s);
    }

    // now print the vec 8 words to a line
    vector<string>::size_type ctr = 0;    // track number of words printed
    // change the for loop to use an iterator, instead of an index
    for (vector<string>::const_iterator i = vec.begin(); i != vec.end(); ++i) {
        cout << *i;   // dereference iterator and print the element
        // check whether it's time to print a newline
        if (++ctr == 8) {
            cout << endl;
            ctr = 0;
        } else
            cout << " ";
    }
    return 0;
}
```

Exercise 3.18: *Write a program to create a* `vector` *with 10 elements. Using an iterator, assign each element a value that is twice its current value.*

Answer 3.18:

```
vector<int> vec(10, 42);   // start out with 10 elements each value 42

// double each value:
// use iterator because we assign to the element value
for (vector<int>::iterator i = vec.begin(); i != vec.end(); ++i)
    // fetch the element, double it and reassign the value
    *i = 2 * *i;   // we could also write *i = 2**i;
```

Exercise 3.19: *Test your previous program by printing the* `vector`*.*

Answer 3.19:

```
// print each value:
// use const_iterator because we only read the element value
for (vector<int>::const_iterator i = vec.begin(); i != vec.end(); ++i)
    cout << *i << endl;
```

Exercises Section 3.4.1

Exercise 3.22: *What happens if we compute* `mid` *as follows:*

```
vector<int>::iterator mid = (vi.begin() + vi.end()) / 2;
```

Answer 3.22: This code is in error: It attempts to add two iterators. We can add an integral value to an iterator but cannot add two iterators. The compiler should generate an error message for this code.

Exercise 3.24: *Consider the sequence 1,2,3,5,8,13,21. Initialize a* `bitset<32>` *object that has a one bit in each position corresponding to a number in this sequence. Alternatively, given an empty* `bitset`*, write a small program to turn on each of the appropriate bits.*

Answer 3.24:

```
#include <iostream>
#include <cstddef>
#include <string>
#include <bitset>
using std::string; using std::bitset; using std::size_t;
using std::cout; using std::endl;
int main()
{
    const size_t bitsz = 32;
    // directly initialize bits1 to the right pattern
    bitset<bitsz> bits(string("00000000001000000010000100101110"));
    // now calculate the bits, rather than directly initializing the bitset
    bitset<bitsz> bits2;          // default bitset; all bits are zero
    /*
     * set bits in bits to correspond to numbers
     * in the fibonacci series: each number is the sum
     * of the preceeding two.
     *
     * On each loop, we set bit corresponding to i
     * and then advance i and j to represent the
     * next pair of numbers in the sequence
     */
    int i = 1, j = 1;
    while (i < bitsz) {
        bits2.set(i);     // set next bit in the sequence
        int k = i + j;
        i = j;
        j = k;
    }
    // check that we set the right bits
    if (bits != bits2)
        cout << "something wrong!" << endl
                     << "\tDirectly setting the pattern yields " << bits << endl
             << "\tComputing the pattern yields " << bits2 << endl;
    return 0;
}
```

Chapter 4

Exercises Section 4.1.1

Exercise 4.1: *Assuming* `get_size` *is a function that takes no arguments and returns an* `int` *value, which of the following definitions are illegal? Explain why.*

```
unsigned buf_size = 1024;

(a) int ia[buf_size];
(b) int ia[get_size()];
(c) int ia[4 * 7 - 14];
(d) char st[11] = "fundamental";
```

Answer 4.1:
(a) Illegal: `buf_size` is not a `const`.
(b) Illegal: `get_size()` is not a constant expression.
(c) Legal: the dimension is a constant expression.
(d) Illegal: The dimension is smaller than the number of supplied initializers. The array must have at least 12 elements—11 for the characters in the string "fundamental" plus one more for the terminating null character.

Exercise 4.2: *What are the values in the following arrays?*

```
string sa[10];
int ia[10];
int main() {
    string sa2[10];
    int    ia2[10];
}
```

Answer 4.2: The `string` arrays have the same element values—each element is initialized using the default `string` construtor, which initializes the element to the empty string.

The `int` arrays differ by scope: The array `ia` is defined at global scope (outside any function) and so is initialized. Each element is initialized to 0. The array `ia2` is local to a function; the element values are uninitialized.

Exercise 4.5: *List some of the drawbacks of using an array instead of a* `vector`*.*

Answer 4.5:

1. The size of an array must be known at compile time.
2. The size of an array is fixed—it is not possible to add or remove elements after the array is defined.
3. It is not possible to copy or assign an array; instead we must do so by writing code to copy or assign the elements from one array into another.

Exercises Section 4.1.2

Exercise 4.6: *This code fragment intends to assign the value of its index to each array element. It contains a number of indexing errors. Identify them.*

```
const size_t array_size = 10;
int ia[array_size];
for (size_t ix = 1; ix <= array_size; ++ix)
      ia[ix] = ix;
```

Answer 4.6:

1. It skips element number 0. That element remains undefined
2. It attempts to assign to an element one past the end of the array. The loop should end before `ix` is equal to `array_size`. An array in C++ runs from element 0 through element size of array minus 1.

One way to correct the loop looks like this:

```
for (size_t ix = 0; ix < array_size; ++ix)
    ia[ix] = ix;
```

Exercise 4.7: *Write the code necessary to assign one array to another. Now, change the code to use* `vectors`*. How might you assign one* `vector` *to another?*

Answer 4.7:

```
#include <vector>
#include <iostream>
#include <cstddef>
using std::vector; using std::cout; using std::endl; using std::size_t;
int main()
{
    // define and initialize an array of 10 elements
    int ia[] = {0,1,2,3,4,5,6,7,8,9};
    // assign ia to another array
    int ia2[10];
    for (size_t i = 0; i != 10; ++i)
        ia2[i] = ia[i];
    // next define a vector of 10 elements
    vector<int> ivec(10, 42);
    // now assign ivec to another vector
    vector<int> ivec2;        // ivec2 has no elements
    // vector supports assignment, no need to write our own loop
    ivec2 = ivec;             // ivec2 now has as many elements as in ivec
    return 0;
}
```

Exercises Section 4.2.2

Exercise 4.12: *Given a pointer,* `p`*, can you determine whether* `p` *points to a valid object? If so, how? If not, why not?*

Answer 4.12: No, it is not possible to know whether a given pointer holds the address of an object. At best, we can know only whether the pointer holds 0, indicating that it does *not* point at any object.

Exercise 4.13: *Why is the first pointer initialization legal and the second illegal?*

```
int i = 42;
void *p = &i;
long *lp = &i;
```

Answer 4.13: Both initializations initialize the pointer from an address. In general, a pointer may be initialized or assigned only from the address of an object of the same type as the pointer.

The exception is type `void*`. A `void*` pointer may hold the address of any non`const` object.

A pointer to `long` may only hold the address of a `long` object; the second initialization attempts to initialize a pointer to `long` from the address of an `int`.

Exercises Section 4.2.3

Exercise 4.16: *What does the following program do?*

```
int i = 42, j = 1024;
int *p1 = &i, *p2 = &j;
*p2 = *p1 * *p2;
*p1 *= *p1;
```

Answer 4.16:

The first statement defines and initailizes two `int` variables.

The second defines and initailizes two pointers that point to these variables.

Assuming this program runs on a machine with `ints` that are larger than 16 bits, the third statement assigns the value 43008 (which is 42 * 1024) to `j`. It executes as follows:

- Dereference `p1` and `p2`, which fetches the current values in `i` and `j`, respectively.
- Multiply the values of `i` and `j`.
- Dereference `p2` in order to assign to it. This statement is effectively the same as `j = i * j`.

The final statement assigns the value 1764 (42 * 42) to `i`:

- The `*=` operator multiplies the left-hand operand by the right-hand operand, storing the result in the left-hand operand.
- The left- and right-hand operands each dereference `p1`, which fetches `i`. The value currently in `i` is 42.

Exercises Section 4.2.4

Exercise 4.18: *Write a program that uses pointers to set the elements in an array of* `ints` *to zero.*

Answer 4.18:

```
#include <iostream>
using std::cout; using std::endl;
int main()
{
    // define and initialize an array of 10 elements
    int ia[] = {0,1,2,3,4,5,6,7,8,9};
        // now use a pointer to assign 0 to each element
        for (int *p = ia; p != ia + 10; ++p)
                *p = 0;
        // now print the array, using subscripts to verify that elements are 0
        for (int n = 0; n != 10; ++n)
                cout << ia[n] << " ";
        cout << endl;
    return 0;
}
```

Exercises Section 4.3

Exercise 4.20: *Which of the following initializations are legal? Explain why.*

```
(a) int i = -1;
(b) const int ic = i;
(c) const int *pic = &ic;
(d) int *const cpi = &ic;
(e) const int *const cpic = &ic;
```

Answer 4.20:

(a) OK: an `int` may hold positive or negative integral values.
(b) OK: we can use a non`const` variable to initialize a `const` variable.
(c) OK: `pic` is a pointer to a `const int` so we can assign the address of the `const` variable `ic` to `pic`.
(d) Error: `cpi` is a `const` pointer. The pointer points to a plain (non`const`) `int`. We cannot use the address of a `const` variable to initialize `cpi` because it is a pointer that could be used to change the value to which it points.
(e) OK: `cpic` is both a `const` pointer—meaning we cannot make `cpic` point to any other object—but it is also a pointer to `const`—meaning we cannot use it to change the value of the object to which it points. Because `cpic` is a pointer to `const` we can initialize it from the address of the `const` variable `ic`.

Exercises Section 4.3

Exercise 4.23: *What does the following program do?*

```
const char ca[] = {'h', 'e', 'l', 'l', 'o'};
const char *cp = ca;
while (*cp) {
    cout << *cp << endl;
    ++cp;
}
```

Answer 4.23: The behavior of this program is undefined. The loop that reads through `cp` assumes that `cp` points to a null-terminated character array. However, `cp` actually point to `ca`, which is not null-terminated.

Although the behavior of this program is undefined, many compilers will generate code that when executed will read memory starting from `ca[0]` until a null character is found.

Exercise 4.25: *Write a program to compare two* `strings`*. Now write a program to compare the value of two C-style character strings.*

Answer 4.25:

```
#include <string>
#include <cstring>
using std::string; using std::strcmp;
int main()
{
    string s1 = "hello", s2 = "world";
    if (s1 == s2)
        { /* do something */ }
    const char *cp1 = "hello";
    const char *cp2 = "world";
    if (strcmp(cp1, cp2) == 0)
        { /* do something */ }
    return 0;
}
```

Exercises Section 4.3.1

Exercise 4.27: *Given the following* `new` *expression, how would you* `delete pa`*?*

```
int *pa = new int[10];
```

Answer 4.27: The memory must be returned using the `delete []` form:

```
delete [] pa;
```

Exercise 4.28: *Write a program to read the standard input and build a* `vector` *of* `int`*s from values that are read. Allocate an array of the same size as the* `vector` *and copy the elements from the* `vector` *into the array.*

Answer 4.28:

```
#include <vector>
#include <iostream>
using std::vector;
using std::cin; using std::cout; using std::endl;
int main()
{
    int i;
    vector<int> vec;
    // read integers from the standard input into vec
    while (cin >> i)
        vec.push_back(i);
    // print what we read
    for (vector<int>::const_iterator it = vec.begin(); it != vec.end(); ++it)
        cout << *it << endl;
    // now copy vec into a dynamically allocated array
    int *p = new int[vec.size()];    // allocate space
    vector<int>::const_iterator it = vec.begin();
    while (it != vec.end())
        // copy element from vec into the array
        // and increment the iterator and pointer
        *p++ = *it++;
    // now print the array contents to check that the copy worked
    int *p2 = p - vec.size(); // get back pointer to first element
    while (p2 != p)
        cout << *p2++ << endl;
    return 0;
}
```

Exercises Section 4.3.2

Exercise 4.32: *Write a program to initialize a* `vector` *from an array of* `int`*s.*

Answer 4.32:

```
#include <vector>
using std::vector;
int main()
{
    // define an array and initialize its elements
    const int arrsize = 10;
    int ia[arrsize] = {0,1,2,3,4,5,6,7,8,9};
    // now initialize a vector from the array
    vector<int> (ia, ia + arrsize);
    return 0;
}
```

Exercise 4.34: *Write a program to read* `string`*s into a* `vector`*. Now, copy that* `vector` *into an array of character pointers. For each element in the* `vector`*, allocate a new character array and copy the data from the* `vector` *element into that character array. Then insert a pointer to the character array into the array of character pointers.*

Answer 4.34:

```
#include <string>
#include <vector>
#include <iostream>
using std::vector; using std::string;
using std::cin; using std::cout; using std::endl;
int main()
{
    // read strings from standard input into a vector
    string s;
    vector<string> vec;
    while (cin >> s)
        vec.push_back(s);
    // define an array of character pointers of the appropriate size
    // arr points to an array of character pointers
    char **arr = new char*[vec.size()];
    // now assign each character pointer to point to a copy
    // of the strings in vec
    for (vector<string>::size_type i = 0; i != vec.size(); ++i) {
        // allocate space to hold the copy plus one for the null
        // *arr is a pointer to a character array, e.g. char*
        *arr = new char[vec[i].size() + 1];
        // do the copy, remember to add 1 to the size to leave space for the null
        strncpy(*arr, vec[i].c_str(), vec[i].size() + 1);
        // increment arr to get the next character pointer
        ++arr;
    }
    char **p = arr - vec.size();    // get pointer to beginning of the array
    while (p != arr)
        cout << *p++ << endl;  // OK: *p is a char*
    return 0;
}
```

Exercises Section 4.4.1

Exercise 4.36: *Rewrite the program to print the contents of the array* `ia` *without using a typedef for the type of the pointer in the outer loop.*

Answer 4.36:

```
// rewrite the loop that printed the contents of the array
// but don't use the int_array typedef
// p points to an int array of with 4 elements
for (int(*p)[4] = ia; p != ia + 3; ++p)
    for (int *q = *p; q != *p + 4; ++q)
         cout << *q << endl;
```

Chapter 5

Exercises Section 5.1

Exercise 5.1: *Parenthesize the following expression to indicate how it is evaluated. Test your answer by compiling the expression and printing its result.*

```
12 / 3 * 4 + 5 * 15 + 24 % 4 / 2
```

Answer 5.1:

```
cout << "parenthesized version: " <<
   ((((12 / 3) * 4) + (5 * 15)) + ((24 % 4) / 2))
<< endl;
```

Exercise 5.3: *Write an expression to determine whether an* `int` *value is even or odd.*

Answer 5.3: Given that `i` is an `int` variable, the expression

```
i % 2
```

returns `0` if `i` is even and `1` if `i` is odd or `-1` (on most implementations) if `i` is odd and negative. We can use this expression as a condition to perform a test to determine whether the value is even or odd.

Exercises Section 5.2

Exercise 5.7: *Write the condition for a* `while` *loop that would read* `int`*s from the standard input and stop when the value read is equal to* `42`*.*

Answer 5.7:

```
// read ints from the standard input until we read the value 42
int i;
while (cin >> i && i != 42) { /* empty */ }
```

Exercise 5.8: *Write an expression that tests four values,* `a`*,* `b`*,* `c`*, and* `d`*, and ensures that* `a` *is greater than* `b`*, which is greater than* `c`*, which is greater than* `d`*.*

Answer 5.8:

```
// test whether a greater than b, b greater than c
// and c greater than d
if (a > b && b > c && c >> d) { /* empty */ }
```

Exercises Section 5.3.1

Exercise 5.9: *Assume the following two definitions:*

```
unsigned long ul1 = 3, ul2 = 7;
```

What is the result of each of the following expressions?

```
(a) ul1 & ul2          (c) ul1 | ul2
(b) ul1 && ul2         (d) ul1 || ul2
```

Answer 5.9: To answer this question, we first observe that the binary representation of `3` is `000...011` and `7` is `000...0111`. So, every bit that is on in `3` is also on in `7`. Which means that:

1. `ul1 & ul2` will result in the bits that are on in both, hence the result is 3

2. `ul1 | ul2` will result in the bits that are on in either, hence the result is 7

3. `ul1 && ul2` is a logical AND test, which is implicitly `ul1 != 0 && ul2 != 0`, both of which are true, so the overall result is the boolean `true`

4. `ul1 || ul2` is the logical OR test, which is implicitly treated as `ul1 != 0` or `ul2 != 0`. Becuase the first condition, `ul1 != 0`, is `true` the overall result is also the boolean `true`.

Exercise 5.12: *Explain what happens in each of the* `if` *tests:*

```
if (42 = i)   // . . .
if (i = 42)   // . . .
```

Answer 5.12: Each condition is an assignment. In the first case, the assignment (illegally) attempts to assign to a literal constant value. This code will be flagged as an error by the compiler. In the second condition, the assignment is legal and the expression evaluates as `true`: The value `42` is assigned to `i` and the result of the assignment is `i`, which when tested as a condition evaluates as `true`.

Exercises Section 5.4.3

Exercise 5.13: *The following assignment is illegal. Why? How would you correct it?*

```
double dval; int ival; int *pi;
dval = ival = pi = 0;
```

Answer 5.13: The assignment is illegal because assignment is right-associative, meaning that this statement is interpreted as assigning 0 to `pi`, then assigning the result of that assigment to `ival`. But `pi` is an `int*` and it is illegal to assign a pointer to `int` to an `int`.

One way to correct the program is:

```
// OK: compiler converts and assigns int value to a double
dval = ival = 0;
// OK: can assign 0 to any pointer type
pi = 0;
```

The assignment from `ival` to `dval` is legal because it is legal to assign an `int` value to a `double`.

Exercises Section 5.5

Exercise 5.16: *Why do you think C++ wasn't named ++C?*

Answer 5.16: The earliest C++ compiler generated C code rather than object code. Hence, C++ "incremented" C by adding support for classes, but returned C as its output; just as prefix increment operator increments its operand but returns the old value of that operand.

Exercises Section 5.6

Exercise 5.19: *Assuming that* `iter` *is a* `vector<string>::iterator`*, indicate which, if any, of the following expressions is legal. Explain the behavior of the legal expressions.*

```
(a) *iter++;             (b) (*iter)++;
(c) *iter.empty()        (d) iter->empty();
(e) ++*iter;             (f) iter++->empty();
```

Answer 5.19:

(a) Legal: dereference `iter` returning a reference to the element to which `iter` refers and increment the iterator to denote the next element.
(b) Illegal: this code dereferences `iter` returning a reference to the element to which `iter` refers and attempts to increment that element. The element to which `iter` refers is of type `string`, but the `string` type does not define an increment operation. Hence, the code is illegal.
(c) Illegal: this expression attempts to run the `empty` member of the object `iter` and then dereference the result returned from `empty`. However, `iter` is a `vector<string>::iterator` and so does not have an `empty` member.

(d) Legal: dereference `iter` returning a reference to the element to which `iter` refers and runs the `empty` member on that element. Because `iter` is a `vector<string>::iterator`, dereferencing `iter` returns a `string`. The `string` type defines an `empty` member. The effect of this expression is to determine whether the `string` referred to by `iter` is empty.
(e) Illegal: As in case (b), this code dereferences `iter` obtaining a reference to the element to which `iter` refers and attempts to increment that element. But the element type is `string` and `string` has no increment operation.
(f) Legal: increment (postfix) `iter` returning a reference to the element to which `iter` originally referred then run the `empty` operation on that element.

Exercises Section 5.7

Exercise 5.20: *Write a program to prompt the user for a pair of numbers and report which is smaller.*

Answer 5.20:

```
#include <iostream>
using std::cin; using std::cout; using std::endl;
int main()
{
    // prompt the user
    cout << "Enter a pair of numbers:" << endl;
    // read the values
    int i, j;
    cin >> i >> j;
    // report which one is smaller
    cout << "The smaller number is: "
         << (i < j ? i : j)
         << endl;
    return 0;
}
```

Exercises Section 5.8

Exercise 5.22: *Write a program to print the size of each of the built-in types.*

Answer 5.22:

```
#include "Sales_item.h"
#include <iostream>
using std::cout; using std::endl;
int main(){
    cout << "bool: " << sizeof(bool) << '\n'
         << "char: " << sizeof(char) << '\n'
         << "wchar_t: " << sizeof(wchar_t) << '\n'
         << "short: " << sizeof(short) << '\n'
         << "int: " << sizeof(int) << '\n'
         << "long: " << sizeof(long) << '\n'
         << "float: " << sizeof(float) << '\n'
         << "double: " << sizeof(double) << '\n'
         << "long double: " << sizeof(long double) << endl;
    return 0;
}
```

Exercises Section 5.10.2

Exercise 5.25: *Using Table 5.4 (p. 170), parenthesize the following expressions to indicate the order in which the operands are grouped:*

```
(a)  ! ptr == ptr->next
(b)  ch = buf[ bp++ ] != '\n'
```

Answer 5.25:

(a) Both `!` and `->` have higher precedence than `==` so this expression compares the boolean result of `!ptr` to the value returned from dereferencing `ptr` and fetching the `next` member from the object to which `ptr` refers:

```
( (!ptr) == (ptr->next) )
```

(b) The subscript operator has the highest precedence in this expression and it has `buf` and `bp++` as its operands. The operand with the next highest precedence is the `!=` operator. Thus, this expression compares the element in `buf` subscripted by `bp++` with the newline character and stores the boolean result of the `!=` comparison in `ch`:

```
(ch = ( (buf[ (bp++) ]) != '\n'))
```

Exercise 5.27: *The following expression fails to compile due to operator precedence. Using Table 5.4 (p. 170), explain why it fails. How would you fix it?*

```
string s = "word";
// add an 's' to the end, if the word doesn't already end in 's'
string pl = s + s[s.size() - 1] == 's' ? "" : "s" ;
```

Answer 5.27: The expression fails to compile because the `==` operator has the lowest precedence of the operators in this expression. Hence, it is interpreted as if we'd written:

```
// compares the result of the addition to the result of conditional operator
// note this code assumes we know that s is not empty
string pl = (s + s[s.size() - 1]) == ('s' ? "" : "s");
// this expression is equivalent to the following separate statements
// note this code assumes we know that s is not empty
string sl = s + s[s.size() - 1]; // addition has higher precedence than ==
bool b1 = 's' ? "" : "s";          // as does the conditional operator
bool b2 s == b1;                   // error: we cannot compare a string and a bool
```

To correct the program to match the intent stated in the comments we can rewrite the expression as:

```
// adds the result of the conditional expression to s
// note this code assumes we know that s is not empty
string pl = s + (s[s.size() - 1] == 's' ? "" : "s");
```

Exercises Section 5.10.3

Exercise 5.29: *Given that* `ptr` *points to a class with an* `int` *member named* `ival`, `vec` *is a* `vector` *holding* `int`s, *and that* `ival`, `jval`, *and* `kval` *are also* `int`s, *explain the behavior of each of these expressions. Which, if any, are likely to be incorrect? Why? How might each be corrected?*

```
(a) ptr->ival != 0          (b) ival != jval < kval
(c) ptr != 0 && *ptr++      (d) ival++ && ival
(e) vec[ival++] <= vec[ival]
```

Answer 5.29:

(a) OK: Assuming `ptr` actually points to an object, this expression uses `ptr` to fetch the `int` member of the object to which `ptr` points and then compares that value with 0.
(b) The code, although legal, is likely to be incorrect. This expression compares the boolean result of comparing `jval < kval` with the value in `ival`.
(c) Legal, but probably incorrect: The expression first checks that `ptr` is not zero. Only if that test succeeds is the right-hand operand of the logical AND executed. That expression dereferences `ptr` to fetch the object to which `ptr` points and then increments `ptr`.

    The only problem with this code is that it dereferences `ptr` and the result of that dereference is used as the right-hand operand of the AND operator. As we've seen, some types such as the IO stream classes allow their objects to be used as a condition. However, not all types support this kind of usage. It's possible that the programmer intended to increment the value to which `ptr` pointed, something like:

```
ptr != 0 && (*ptr).ival++
```

    or that the dereference is a mistake:

```
ptr != 0 && ptr++
```

    without more information it is hard to understand the intent.
(d) Legal, because the AND operator guarantees the order in which its operands are evaluated. This expression first evaluates the left-hand operand, incrementing `ival`. The expression uses the postfix increment operator, so the result of the increment expression is the old value of `ival`. If that value was nonzero, then the right-hand operand is evaluated, which returns the current (incremented) value of `ival`.
(e) Undefined: Both operands to the `<=` operator use the same object and one of the operands changes the value of that object. Unlike the AND operator, the `<=` operator makes no guarantees as to the order in which its operands are evaluated. Because we cannot know which operand is evaluated first, it is impossible to know what values are used for the subscripts in the right- and left-hand expressions.

Exercises Section 5.11

Exercise 5.30: *Which of the following, if any, are illegal or in error?*

```
(a) vector<string> svec(10);
(b) vector<string> *pvec1 = new vector<string>(10);
(c) vector<string> **pvec2 = new vector<string>[10];
(d) vector<string> *pv1 = &svec;
(e) vector<string> *pv2 = pvec1;

(f) delete svec;
(g) delete pvec1;
(h) delete [] pvec2;
(i) delete pv1;
(j) delete pv2;
```

Answer 5.30:

(a) Legal: defines a `vector<string>` object with 10 elements; each element is the empty string.
(b) Legal: dynamically allocates a `vector<string>` object with 10 elements; each element is the empty string. Stores a pointer to the allocated `vector` in `pvec1`.
(c) Illegal: The `new` expression dynamically allocates an array of size 10 each of whose elements is an empty `vector<string>`. That `new` returns a pointer to the first element of the array. The type of the elements is `vector<string>`, so the pointer is a `vector<string>*`. The type of `pvec2` is a pointer to a pointer to a `vector<string>`, which is not the same type as the type returned by this `new` expression. Assuming the `new` correctly reflects the programmer's intent, then the correct declaration would be:

```
// allocate an array of vectors
vector<string> *pvec2 = new vector<string>[10];
```

If the programmer intended to allocate an array of pointers to `vector<string>` then the declaration could be written as:

```
// allocate an array of pointers to vectors
vector<string> **pvec2 = new vector<string>*[10];
```

(d) Legal: assigns the address of a `vector<string>` to a pointer of the same (e.g. `vector<string>*`) type.
(e) Legal: assigns two pointers. Both the left- and right-hand operands are pointers of the same type: They're both pointers to `vector<string>`.
(f) Illegal: This `delete` attempts to delete an object, not a pointer.
(g) Ok: `deletes` a pointer to a dynamically allocated object.
(h) OK, assuming the declaration of `pvec2` is corrected and points to a dynamically allocated array.
(i) Illegal, but unlikely to be detected: `pv1` holds the address of `svec`, which was not dynamically allocated. Deleting a pointer to non-dynamically allocated object is an error. However, the compiler has no way to know which object `pv1` points to and so is unlikely to detect this error.
(j) OK: `pv2` points to a dynamically allocated array. However, given the assignments in the first half of this exercise, only one of `delete pv2` and `delete pvec1` may be executed. That is, it is illegal to `delete` the same dynamically allocated memory twice.

Exercises Section 5.12.7

Exercise 5.32: *Given the following set of definitions,*

```
char cval;     int ival;    unsigned int ui;
float fval;                 double dval;
```

identify the implicit type conversions, if any, taking place:

```
(a) cval = 'a' + 3;       (b) fval = ui - ival * 1.0;
(c) dval = ui * fval;     (d) cval = ival + fval + dval;
```

Answer 5.32:

(a) The `char 'a'` is promoted to `int` in order to add it to the literal `3`. The result of the addition is an `int`, which is then converted to `char` in order to assign the value to `cval`.
(b) The `int ival` is converted to `double` in order to multiply it by `1.0`. That result is a `double`, which is a larger type then `unsigned int`. Hence, `ui` is converted to `double` to do the subtraction. The result of the subtraction is a `double`, which is converted to `float` in order to assign it to `fval`.
(c) `ui` is converted to `float` in order to do the multiplication with `fval`. The result of the multiplication is a `float`, which is converted to `double` in order to assign it to `dval`.
(d) Because addition is left-associative, the `int` must be converted to `float` in order to add it to `fval`. That result, which is of type `float`, is added to `dval` so the result must be converted to `double`. The result of the addition to `dval` is a `double`, which is converted to `char` in order to assign the result to `cval`.

Chapter 6

Exercises Section 6.3

Exercise 6.3: *Use the comma operator (Section 5.9, p. 168) to rewrite the* `else` *branch in the* `while` *loop from the bookstore problem so that it no longer requires a block. Explain whether this rewrite improves or diminishes the readability of this code.*

Answer 6.3: We could rewrite the two statements in the block as a single statement using a comma as follows:

```
std::cout << total << std::endl, total = trans;
```

However, doing so is likely to be confusing to readers. Moreover, it is worth noting that this rewrite works only because the comma operator guarantees the order of evaluation of its operands. We are guaranteed that the output expressions are evaluated before the assignment.

Exercises Section 6.5.1

Exercise 6.5: *Correct each of the following:*

```
(a) if (ival1 != ival2)
         ival1 = ival2
    else ival1 = ival2 = 0;

(b) if (ival < minval)
         minval = ival;   // remember new minimum
         occurs = 1;      // reset occurrence counter

(c) if (int ival = get_value())
         cout << "ival = " << ival << endl;
    if (!ival)
         cout << "ival = 0\n";

(d) if (ival = 0)
         ival = get_value();
```

Answer 6.5:

(a) The statement that follows the `if` test is missing a semicolon. It should be:

```
if (ival1 != ival2)
     ival1 = ival2;   // note, semicolon added
else ival1 = ival2 = 0;
```

(b) The indentation indicates, and the program presented in this section also requires, that the statements that follow the `if` should be executed as a block. Curly braces are needed:

```
if (ival < minval) { // note open curly added
     minval = ival;
     occurs = 1;
}                    // note close curly added
```

(c) The variable `ival` is defined local to the `if`. In order to access that value in the second `if`, we'd have to define `ival` outside either `if`:

```
int ival = get_value();          // moved definition of ival
if (ival)
     cout << "ival = " << ival << endl;
if (!ival)
     cout << "ival = 0\n";
```

(d) This `if` is strictly speaking legal, but likely to be in error. The condition in the `if` assigns 0 to `ival` and then tests to see whether `ival` is 0 or not. The assignment means that the `if` test will always fail—remembering that when an `int` value used as a condition, 0 converts to `false`. It is more likely that the programmer intended to compare `ival` to 0:

```
if (ival == 0)
     ival = get_value();
```

which might more succinctly (and with less chance for error) be rewritten as:

```
if (!ival)
     ival = get_value();
```

Exercises Section 6.6.5

Exercise 6.7: *There is one problem with our vowel-counting program as we've implemented it: It doesn't count capital letters as vowels. Write a program that counts both lower- and uppercase letters as the appropriate vowel—that is, your program should count both* `'a'` *and* `'A'` *as part of* `aCnt`*, and so forth.*

Answer 6.7:

```
char ch;
//  initialize counters for each vowel
int aCnt = 0, eCnt = 0, iCnt = 0,
    oCnt = 0, uCnt = 0;
while (cin >> ch) {
    //  if ch is a vowel, increment the appropriate counter
    switch (ch) {
        case 'A':
        case 'a':
            ++aCnt;
            break;
        case 'E':
        case 'e':
            ++eCnt;
            break;
        case 'I':
        case 'i':
            ++iCnt;
            break;
        case 'O':
        case 'o':
            ++oCnt;
        break;
        case 'U':
        case 'u':
            ++uCnt;
            break;
    }
}
//  print results
cout << "Number of vowel a: \t" << aCnt << '\n'
     << "Number of vowel e: \t" << eCnt << '\n'
     << "Number of vowel i: \t" << iCnt << '\n'
     << "Number of vowel o: \t" << oCnt << '\n'
     << "Number of vowel u: \t" << uCnt << endl;
```

Exercise 6.10: *Each of the programs in the highlighted text on page 206 contains a common programming error. Identify and correct each error.*

Answer 6.10:

(a) This code incorrectly omits `break` statements after the case labels that handle processing for characters `'a'` and `'e'`. When an `'a'` is seen, all three counters are incremented. To fix the problem, rewrite the code inserting a `break` after each case label:

```
switch (ival) {
    case 'a': aCnt++; break;   // break statement added
    case 'e': eCnt++; break;   // break statement added
    default: iouCnt++;
}
```

(b) The definition of `ix` after the first case label is illegal. Variables in a `switch` may be defined only after the last label or inside a block. Because both the `default` case and the initial case need access to this variable, one solution is to define `ix` before the `switch` statement:

```
int ix = get_value();   // definition moved outside the switch statement
switch (ival) {
    default:              // the default label does not have to be last
        ix = ivec.size()-1; // no break
    case 1:
        ivec[ ix ] = ival;
        break;
}
```

An even better rewrite would be to realize that this code is not well-suited to a `switch` statement and rewrite it more directly using an `if`:

```
int ix = get_value();
if (ival != 1)
    ix = ivec.size() - 1;
ivec[ix] = ival;
```

(c) The case labels are invalid. Each case label may contain only a single value. The fix is to repeat the keyword `case` in front of each value:

```
switch (ival) {
    case 1: case 3: case 5: case 7: case 9:
        oddcnt++;
        break;
    case 2: case 4: case 6: case 8: case 10:
        evencnt++;
        break;
}
```

(d) Case labels must be constant integral expressions. This program uses non`const int` variables as the labels. We could fix the program by introducing new `const` variables to hold these values and/or by changing the existing variables to be `const`:

```
// changed ival, jval, and kval to be const
const int ival=512 jval=1024, kval=4096;
int bufsize;
// ...
switch(swt) {
    case ival:
        bufsize = ival * sizeof(int);
        break;
    case jval:
        bufsize = jval * sizeof(int);
        break;
    case kval:
        bufsize = kval * sizeof(int);
        break;
}
```

Exercises Section 6.7

Exercise 6.11: *Explain each of the following loops. Correct any problems you detect.*

```
(a) string bufString, word;
    while (cin >> bufString >> word) { /* ... */ }

(b) while (vector<int>::iterator iter != ivec.end())
    { /* ... */ }

(c) while (ptr = 0)
        ptr = find_a_value();

(d) while (bool status = find(word))
    { word = get_next_word(); }
    if (!status)
         cout << "Did not find any words\n";
```

Answer 6.11:

(a) The condition in the `while` reads two `string` values from `cin` storing what is read in `bufString` and `word`. The loop continues until `cin` hits end-of-file or encounters some other input error.

(b) The condition in the `while` as written is in error—the condition starts out like a declaration but then uses the != operator. To fix the code, we can guess that the loop probably is intended to read through the `vector` to which `iter` is bound. The loop probably should be rewritten as:

```
// initialize iter to refer to first element in ivec
vector<int>::iterator iter = ivec.begin();

// change condition to compare current value in iter to ivec.end()
while (iter != ivec.end())
{ /* ... */ }
```

Note that the loop body must increment `iter` or else the loop will run indefinitely.

(c) The condition assigns zero to `ptr` and then tests the value of `ptr`. That value will be 0 and the loop will never execute. The condition almost surely was intended to compare `ptr` with 0:

```
while (ptr == 0)
    ptr = find_a_value();
```

Or written more succinctly as:

```
while (!ptr)
    ptr = find_a_value();
```

(d) This program invalidly attempts to access the `bool` variable `status` outside the `while` loop in which it is defined. It might be corrected as:

```
bool status = false;          // moved definition of status outside the loop
while (find(word)) {
    word = get_next_word();
    status = true;     // we found a word we wanted
}
if (!status)
     cout << "Did not find any words\n";
```

Exercises Section 6.8.2

Exercise 6.16: *Given two* `vectors` *of* `ints`*, write a program to determine whether one* `vectors` *is a prefix of the other. For* `vectors` *of unequal length, compare the number of elements of the smaller* `vector`*. For example, given the* `vectors` *(0,1,1,2) and (0,1,1,2,3,5,8), your program should return* `true`*.*

Answer 6.16:

```
#include <vector>
#include <iostream>
using std::vector; using std::cin; using std::cout; using std::endl;
int main()
{
    vector<int> v1, v2;
    // read 10 ints into v1
    for (vector<int>::size_type i = 0; i != 10; ++i) {
        int n;
        cin >> n;
        v1.push_back(n);
    }

    // read 4 ints into v2
    for (vector<int>::size_type i = 0; i != 4; ++i) {
        int n;
        cin >> n;
        v2.push_back(n);
    }
    // we need to stop looking once we've exhausted the shorter of the vectors
    vector<int>::size_type sz = v1.size() < v2.size()
                                          ? v1.size() : v2.size();
    // now compare the vectors to see if v2 is a prefix of v1
    vector<int>::size_type i = 0; // i is needed outside the for
    for ( /* empty */; i != sz && v1[i] == v2[i]; ++i)
              { /* empty, work is done in the condition */ }
    if (i == sz)
        cout << "true" << endl;
    else
        cout << "false" << endl;
    return 0;
}
```

Exercises Section 6.9

Exercise 6.18: *Write a small program that requests two* `string`*s from the user and reports which* `string` *is lexicographically less than the other (that is, comes before the other alphabetically). Continue to solicit the user until the user requests to quit. Use the* `string` *type, the* `string` *less-than operator, and a* `do while` *loop.*

Answer 6.18: We can write the program as described in the exercise as follows:

```
#include <string>
#include <iostream>
using std::string; using std::cin; using std::cout; using std::endl;
int main()
{
    // must define these variables outside the loop so that the
    // condition in the while can check whether user asked to quit
    string s1, s2;
    do {
        // prompt user and read two strings
        cout << "Enter two strings to compare or 'q' to quit:" << endl;
        cin >> s1 >> s2;
        // if we got input, then report which is smaller
        if (cin && s1 != "q")
            if (s1 != s2)
```

```
                cout << (s1 < s2 ? s1 : s2)
                     << " is smaller" << endl;
            else
                cout << "Both strings are the same" << endl;
    } while (cin && s1 != "q");
    return 0;
}
```

However, this program is not a particularly good match for a `do while` loop. The problem is that we must repeat the test to determine whether the user has finished inside the loop and inside the `while` condition.

We might more succinctly write the program using a `while`:

```
int main()
{
    string s1, s2;
    // condition prompts the user and reads two strings, checking
    // whether the user entered a |q| to quit or entered end-of-file
    while (cout << "Enter two strings to compare or 'q' to quit:" << endl &&
           cin >> s1 && s1 != "q" && cin >> s2) {
        // ok: s1 and s2 contain values entered by the user
        // compare them and report which is smaller
        if (s1 != s2)
            cout << (s1 < s2 ? s1 : s2)
                 << " is smaller" << endl;
        else
            cout << "Both strings are the same" << endl;
    }
    return 0;
}
```

Exercises Section 6.10

Exercise 6.19: *The first program in this section could be written more succinctly. In fact, its action could be contained entirely in the condition in the* `while`*. Rewrite the loop so that it has an empty body and does the work of finding the element in the condition.*

Answer 6.19: The wording of this exercise is too strong: The program can be rewritten easily to avoid the `break` and the condition can do the work of finding the element. However, without unnecessary and useless contortions, the loop cannot be written with an empty body. The body is needed to do the increment to the iterator. Assuming the exercise is corrected to allow the increment inside the loop, we might write the loop as:

```
vector<int>::iterator iter = vec.begin();
while (iter != vec.end() && value != *iter)
{
    // not found yet, look at the next element
    ++iter;
}
if (iter != vec.end()) // did we exit the loop because we found the element?
    // continue processing
```

Exercises Section 6.12

Exercise 6.22: *The last example in this section that jumped back to* `begin` *could be better written using a loop. Rewrite the code to eliminate the* `goto`*.*

Answer 6.22:

```
//  rewrite loop to eliminate need for a goto
int sz;
while ((sz = get_size()) <= 0)
//  ...
```

Exercises Section 6.13.2

Exercise 6.24: *Revise your program to catch this exception and print a message.*

Answer 6.24:

```
#include <iostream>
#include <string>
#include <bitset>
#include <stdexcept>
#include <cstddef>
using std::cout; using std::endl; using std::size_t;
using std::bitset; using std::string; using std::overflow_error;
int main()
{
    //  number of bits in an unsigned long assuming 8 bits in a byte
    const size_t ul_sz = sizeof(unsigned long) * 8;

    //  first call to_ulong in a situation where it should not overflow
    bitset<ul_sz> bits1;   //  all bits are zero
    bits1.set();           //  turn all bits on
    try {
        bits1.to_ulong();
    } catch (overflow_error) {
        cout << "case 1: to_ulong overflow!" << endl;
    }
    //  twice as many bits as fit in an unsigned long, all zero
    bitset<2 * ul_sz> bits2;
    bits2.set();  //  set all the bits to 1

    try {
        bits2.to_ulong();
    } catch (overflow_error) {
        cout << "case 2: to_ulong overflow!" << endl;
    }
    return 0;
}
```

When executed this program should print

```
case 2: to_ulong overflow!
```

Exercises Section 6.14

Exercise 6.27: *Explain this loop:*

```
string s;
while (cin >> s && s != sought) { }  //  empty body
assert(cin);
//  process s
```

Answer 6.27: The loop reads `strings` from `cin` until it reads a value that is equal to the `string` in `sought`. The loop will be exited either if `cin` hits end-of-file (or encounters some other input error) or if the value `sought` is found.

The program evidently assumes that `sought` will be found. It tests this assumption in the `assert`: If the loop exits because of end-of-file (or other input error), then `cin` will compare as `false` in the condition in the `assert`. If the `assert` fails, then the program will be aborted.

Chapter 7

Exercises Section 7.1.2

Exercise 7.2: *Indicate which of the following functions are in error and why. Suggest how you might correct the problems.*

```
(a) int f() {
        string s;
        // ...
        return s;
    }
(b) f2(int i) { /* ... */ }
(c) int calc(int v1, int v1) /* ... */ }
(d) double square(double x) return x * x;
```

Answer 7.2:

(a) Error: The return type of the function and the type of the value in the `return` statement do not match. Probably the function should be defined to return a `string`:

```
string f() {
    string s;
    // ...
    return s;
}
```

(b) Error: The function does not define a return type. Assuming the function has no return value, we could correct it by making the return type `void`:

```
void f2(int i) { /* ... */ }
```

(c) Error: The function uses the same parameter name to refer to more than one parameter. It is also missing the open curly that starts the function body. To fix it we must give each parameter a unique name and add an open curly:

```
int calc(int v1, int v2) { /* ... */ }
```

(d) Error: This function is missing the curly braces that should enclose the function body. The fix is to enclose the `return` in braces:

```
double square(double x) { return x * x; }
```

Exercise 7.4: *Write a program to return the absolute value of its parameter.*

Answer 7.4:

```
// function to return absolute value of an int
// NB: the <cmath> header defines a similar function named abs
//     that can be used on any of the arithmetic types
int absval(int i)
{
    if (i < 0)
        return -i;
```

```
        else
            return i;
}
```

Exercises Section 7.2.1

Exercise 7.5: *Write a function that takes an* `int` *and a pointer to an* `int` *and returns the larger of the* `int` *value of the value to which the pointer points. What type should you use for the pointer?*

Answer 7.5: The parameters to the function should be `int` and `const int*`. The return type is `int`:

```
int comparevals(int i, const int *p)
{
    return i > *p ? i : *p;
}
```

Exercises Section 7.2.2

Exercise 7.7: *Explain the difference in the following two parameter declarations:*

```
void f(T);
void f(T&);
```

Answer 7.7: The first parameter is a plain, nonreference type. Arguments passed to this parameter will be copied. The second parameter is a reference. This parameter is just another name for the object passed as the argument to this function. The argument must be an lvalue.

Exercise 7.9: *Change the declaration of* `occurs` *in the parameter list of* `find_val` *(defined on page 234) to be a nonreference argument type and rerun the program. How does the behavior of the program change?*

Answer 7.9: We would declare `occurs` as a nonreference as follows:

```
#include <vector>
std::vector<int>::const_iterator find_val(
    std::vector<int>::const_iterator beg,   // first element
    std::vector<int>::const_iterator end,   // one past last element
    int value,                              // the value we want
    std::vector<int>::size_type occurs);    // number of times it occurs
```

However, making `occurs` a nonreference would break the program. The code inside the function would still increment `occurs` each time the sought value was found but those increments would not be reflected in the value of the argument passed to `occurs`. After the return from `find_val`, the argument bound to `occurs` would be unchanged.

Exercise 7.10: *The following program, although legal, is less useful than it might be. Identify and correct the limitation on this program:*

```
bool test(string& s) { return s.empty(); }
```

Answer 7.10: Even though the function does not change the value of its parameter, the parameter is defined as a non`const` reference. Making the parameter a non`const` reference means that only `string` lvalue arguments may be passed. Values—such as a string literal or the result of adding two `strings`—cannot be passed. We can make the program more general by making the parameter a `const` reference:

```
bool test(const string& s) { return s.empty(); }
```

Exercises Section 7.2.5

Exercise 7.14: *Write a program to sum the elements in a* `vector<double>`*.*

Answer 7.14:

```
double vecsum(vector<double>::iterator beg,
              vector<double>::iterator end)
{
    double retval = 0;
    while (beg != end)
        retval += *beg++;
    return retval;
}
```

Exercises Section 7.2.6

Exercise 7.15: *Write a* `main` *function that takes two values as arguments and print their sum.*

Answer 7.15: This program relies on a C library function, named `strtod`, which returns the numeric equivalent of its first character string argument. The `strtod` takes two pointers to C-style character arrays. The first points to the string we want to convert and the second is a pointer to a character pointer. When that argument is zero, it is ignored.

```
#include <iostream>
#include <cstring>
using std::cerr; using std::cout; using std::endl; using std::strtod;
int main(int argc, char *argv[])
{
    if (argc != 3)
        cerr << "wrong number of arguments: "
             << argc - 1 << " expected, got " << 2 << endl;
    cout << "Sum of " << argv[1] << " and " << argv[2]
         << " is " << strtod(argv[1], 0) + strtod(argv[2], 0) << endl;
    return 0;
}
```

Please note, printings after the 3rd printing will be corrected to avoid the need for `strtod` by using `strings` instead of `int` arguments to `main`.

Exercises Section 7.3.2

Exercise 7.18: *What potential run-time problem does the following function have?*

```
string &processText() {
    string text;
    while (cin >> text) { /* ... */ }
    // ....
    return text;
}
```

Answer 7.18: This function is likely to fail at runtime because it returns a reference to a local variable. The storage used by that variable is no longer around after the function returns. The reference that is returned refers to memory that is no longer valid.

Exercise 7.19: *Indicate whether the following program is legal. If so, explain what it does; if not, make it legal and then explain it:*

```
int &get(int *arry, int index) { return arry[index]; }
int main() {
    int ia[10];
    for (int i = 0; i != 10; ++i)
        get(ia, i) = 0;
}
```

Answer 7.19: The program is legal. The `get` function takes a pointer to an array of `int` and an index. It returns a reference to the element in the array at the position indicated by the index. The `main` function defines an (uninitialized) array of 10 `int`s and calls `get`, which returns a reference to the indicated element in the array. It then assigns 0 to the element that was returned.

Exercises Section 7.3.3

Exercise 7.21: *What would happen if the stopping condition in* `factorial` *were:*

```
if (val != 0)
```

Answer 7.21: If the stopping condition were `val != 0` instead of `val > 1` then the program would fail if given an argument with a nonpositive value. If the argument is 0 or a negative number then the recursion will never terminate. The likely runtime behavior is to crash after memory is exhausted.

Exercises Section 7.4

Exercise 7.22: *Write the prototypes for each of the following functions:*

(a) *A function named* `compare` *with two parameters that are references to a class named* `matrix` *and with a return value of type* `bool`*.*
(b) *A function named* `change_val` *that returns a* `vector<int>` *iterator and takes two parameters: one is an* `int` *and the other is an iterator for a* `vector<int>`*.*

Hint: When you write these prototypes, use the name of the function as an indicator as to what the function does. How does this hint affect the types you use?

Answer 7.22:

(a) Because the function is named `compare` and returns `bool` we can infer that it reads and compares elements in its two `matrix` arguments. We'd expect the function to read but not write those elements. We also expect that a `matrix` is likely to be a large data structure, so we'd like to avoid function call overhead. Given this reasoning, the parameters should be `const` references.

```
bool compare(const matrix&, const matrix&);
```

(b) Given the function name and parameters, we might infer that this function sets the element referred to by its iterator parameter to the value of its `int` parameter. Both `int` and `vector<int>::iterator` are simple types, so there is no need to avoid copying the values; we can pass them as simple, nonreference types. Because the function is likely to change the value to which the iterator refers we pass the parameter as `vector<int>::iterator` and not `vector<int>::const_iterator`. If we passed the `const_iterator` type, then the function would not be able to change the element to which the argument referred.

```
vector<int>::iterator change_val(int, vector<int>::iterator);
```

Exercises Section 7.4.1

Exercise 7.25: *Given the following function declarations and calls, which, if any, of the calls are illegal? Why? Which, if any, are legal but unlikely to match the programmer's intent? Why?*

```
// declarations
char *init(int ht, int wd = 80, char bckgrnd = ' ');

(a) init();
(b) init(24,10);
(c) init(14, '*');
```

Answer 7.25:

(a) Illegal: `init` requires at least one argument, representing the height.
(b) Legal and likely to be correct. The arguments appear to set the height and width and the call will use the default value for background.
(c) Legal, but likely to be incorrect. The arguments are bound to `ht` and `wd`, implying that the height parameter is set to `14` and the width parameter to the numeric value of the character `'*'`. It is more likely that the user intended to pass `'*'` as the background character. To do so, arguments must be supplied for all three parameters.

Exercises Section 7.5.2

Exercise 7.28: *Write a function that returns 0 when it is first called and then generates numbers in sequence each time it is called again.*

Answer 7.28:

```
size_t callcnt()
{
    // value of ctr will be preserved across calls to callcnt
    static size_t ctr = 0;
    // uses postfix increment so the unincremented value of ctr is returned
    return ctr++;
}
```

Exercises Section 7.6

Exercise 7.30: *Rewrite the* `isShorter` *function from page 235 as an* `inline` *function.*

Answer 7.30:

```
// compare the length of two strings
inline
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
```

Exercises Section 7.7.4

Exercise 7.32: *Write a header file to contain your version of the* `Sales_item` *class. Use ordinary C++ conventions to name the header and any associated file needed to hold non-*`inline` *functions defined outside the class.*

Answer 7.32: The header file, `Sales_item.h` contains the class definition:

```
#ifndef SALESITEM_H
```

```
#define SALESITEM_H
// Definition of Sales_item class and related functions goes here
#include <iostream>
#include <string>
class Sales_item {
public:
    // default constructor needed to initialize members of built-in type
    Sales_item(): units_sold(0), revenue(0.0) { }
    // constructor to initialize from a string
    Sales_item(const std::string &book):
              isbn(book), units_sold(0), revenue(0.0) { }
    // operations on Sales_item objects
    double avg_price() const { return revenue / units_sold; }
    bool same_isbn(const Sales_item &rhs) const
        { return isbn == rhs.isbn; }
    // named operations to read and write Sales_item objects
    void read(std::istream&);
    void write(std::ostream&) const;
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};
#endif
```

The source file `Sales_item.cc` contains definitions for the input and output functions:

```
#include "Sales_item.h"
#include <iostream>
using std::istream; using std::ostream;
using std::cin; using std::cout; using std::endl;
void Sales_item::read(istream& in)
{
    // read transaction values
    double price;
    in >> isbn >> units_sold >> price;
    // calculate revenue
    revenue = units_sold * price;
}
void Sales_item::write(ostream& out) const
{
    out << isbn << "\t" << units_sold << "\t"
        << revenue << "\t" <<  avg_price();
}
```

Exercises Section 7.8.1

Exercise 7.34: *Define a set of overloaded functions named* `error` *that would match the following calls:*

```
int index, upperBound;
char selectVal;
// ...
error("Subscript out of bounds: ", index, upperBound);
error("Division by zero");
error("Invalid selection", selectVal);
```

Answer 7.34:

```
void error(const string&, int, int);
void error(const string&);
void error(const string&, char);
```

Exercises Section 7.8.3

Exercise 7.37: *Given the declarations for* `f`*, determine whether the following calls are legal. For each call list the viable functions, if any. If the call is illegal, indicate whether there is no match or why the call is ambiguous. If the call is legal, indicate which function is the best match.*

```
(a) f(2.56, 42);
(b) f(42);
(c) f(42, 0);
(d) f(2.56, 3.14);
```

Answer 7.37:

(a) `f(2.56, 42)` is ambiguous. The viable functions are those that take two arguments: `f(int, int)` and `f(double, double)`. The first argument is an exact match for `f(double, double)` whereas the second argument is an exact match for `f(int, int)`.
(b) `f(42)` calls `f(int)`. Because the function `f(double, double)` has a default argument, it can be called with a single argument. Therefore, there are two viable functions for this call: `f(int)` and `f(double, double)`. The argument is an exact match for `f(int)` but would require a conversion in order to call `f(double, double)`. Hence the best match for this call is `f(int)`.
(c) `f(42, 0)` calls `f(int, int)`. The viable functions are those that can be called with two arguments: `f(double, double)` and `f(int, int)`. In this call both arguments are exact matches for `f(int, int)` but would require conversions to call `f(double, double)`. Therefore, the best match is `f(int, int)`.
(d) `f(2.56, 3.14)` calls `f(double, double)`. As in the previous example, the viable functions are those that can be called with two arguments: `f(double, double)` and `f(int, int)`. In this call both arguments are exact matches for `f(double, double)` but would require conversions to call `f(int, int)`. Therefore, the best match is `f(double, double)`.

Exercises Section 7.8.4

Exercise 7.39: *Explain the effect of the second declaration in each one of the following sets of declarations. Indicate which, if any, are illegal.*

```
(a) int calc(int, int);
    int calc(const int&, const int&);

(b) int calc(char*, char*);
    int calc(const char*, const char*);

(c) int calc(char*, char*);
    int calc(char* const, char* const);
```

Answer 7.39:

(a) The second declaration is a separate function. However, these functions can be called only if the caller explicitly indicates which function should be invoked. For example, a call such as `f(i, j)` where `i` and `j` are both `int`s will be ambiguous. The `int` variables `i` and `j` are an exact match for either `int` or `const int&`. In order to call either function the caller would have to include an explicit cast. Because a cast would be necessary this kind of overloading, while legal, is a very bad idea.
(b) The second declaration defines a separate, independent function. The parameters to these functions differ as to whether they can be called with a plain `char` pointer or a pointer to `const char`.

(c) The second declaration is a redeclaration of the first. There is only one function and that function takes a pointer to plain `char`. The pointer itself might be `const` or non`const`.

Chapter 8

Exercises Section 8.1

Exercise 8.1: *Assuming* `os` *is an* `ofstream`*, what does the following program do?*

```
os << "Goodbye!" << endl;
```

What if `os` *is an* `ostringstream`*? What if* `os` *is an* `ifstream`*?*

Answer 8.1: The program prints the charater string `Goodbye!` to the stream to which `os` is attached. It does the same thing if `os` is an `ostringstream`. If `os` is an `ifstream`, the code is in error.

Exercise 8.2: *The following declaration is in error. Identify and correct the problem(s):*

```
ostream print(ostream os);
```

Answer 8.2: This declaration says that the parameter and return type are copied. However, it is not possible to copy an `ostream` object. To correct the program, the `ostream` parameter and return type should be declared as references:

```
ostream& print(ostream& os);
```

Exercises Section 8.2

Exercise 8.3: *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 and return the stream.*

Answer 8.3:

```
#include <iostream>
#include <string>
using std::string; using std::istream; using std::cout; using std::endl;
istream &readfile(istream &in)
{
    string s;
    // read the istream we are passed until an error is encountered
    while (getline(in, s))
        cout << s << endl;  // print what was read to the standard output
    in.clear();  // clear the stream so that it is in a valid state
    return in;   // return the stream
}
```

Exercise 8.4: *Test your function by calling it passing* `cin` *as an argument.*

Answer 8.4:

```
#include <iostream>
using std::cin; using std::cout; using std::endl; using std::istream;
istream &readfile(istream &);
int main()
{
```

```
    readfile(cin);
    if (cin) {
        cout << "OK: status reset to valid" << endl;
        return 0;  // indicate success
    } else {
        cout << "OOPS: something's wrong, stream is not valid" << endl;
        if (cin.eof()) cout << "eof wasn't cleared" << endl;
        if (cin.fail()) cout << "fail bit is still set" << endl;
        if (cin.bad()) cout << "bad bit is still set" << endl;
        return -1; // indicate failure
    }
}
```

Exercises Section 8.4.1

Exercise 8.6: *Because* `ifstream` *inherits from* `istream`*, we can pass an* `ifstream` *object to a function that takes a reference to an* `istream`*. Use the function you wrote for the first exercise in Section 8.2 (p. 291) to read a named file.*

Answer 8.6:

```
#include <iostream>
#include <fstream>
using std::ifstream; using std::istream; using std::cout; using std::endl;
istream &readfile(istream&);
int main()
{
    ifstream file("readfile.cc");    // open a file
    readfile(file);                  // call readfile to read an ifstream
    if (file) {
        cout << "OK: status reset to valid" << endl;
        return 0;  // indicate success
    } else {
        cout << "OOPS: something's wrong, stream is not valid" << endl;
        if (file.eof()) cout << "eof wasn't cleared" << endl;
        if (file.fail()) cout << "fail bit is still set" << endl;
        if (file.bad()) cout << "bad bit is still set" << endl;
        return -1; // indicate failure
    }
}
```

Exercise 8.8: *The programs in the previous exercise can be written without using a* `continue` *statement. Write the program with and without using a* `continue`*.*

Answer 8.8: The most direct way to replace the `continue` is to realize that it wasn't strictly speaking ever necessary. As written, the inner `while` loop already correctly handles the case that for some reason the `input` file is invalid. If the file is invalid, then the `while` that reads and processes the file fails on the first iteration. Because `input` is invalid, the attempt to read `input` will fail. The condition in the `while` will test `input`, which will fail and the loop will be exited without ever executing the body of the `while`:

```
// for each file in the vector
while (it != files.end()) {
    ifstream input(it->c_str());  // open the file;
    if (!input)
        cerr << "something's wrong: open failed for " << it->c_str() << endl;
    // if the file is ok, the while will read and "process" the input
```

```
        // if the open failed, the while will exit without ever executing the body
        while(input >> s)               // do the work on this file
            process(s);
        ++it;                           // increment iterator to get next file
    }
```

The second loop is only slightly more complicated. Again the inner `while` that calls `process` is safe, regardless of whether the `open` succeeded. However, we do have to check that the `open` succeeded before calling `close`:

```
    ifstream input;
    vector<string>::const_iterator it = files.begin();
    // for each file in the vector
    while (it != files.end()) {
        input.open(it->c_str());   // open the file
        if (!input)
            cerr << "something's wrong: open failed for " << it->c_str() << endl;
        // if the file is ok, the while will read and ''process'' the input
        // if the open failed, the while will exit without ever executing the body
        while(input >> s) // do the work on this file
            process(s);
        if (input.is_open())
            input.close();      // if we succeeded in opening the file, close it
        input.clear();          // reset state to ok
        ++it;                   // increment iterator to get next file
    }
```

Exercise 8.10: *Rewrite the previous program to store each word in a separate element.*

Answer 8.10:

```
#include <fstream>
#include <vector>
#include <string>
#include <stdexcept>
using std::ifstream; using std::vector; using std::string;
using std::invalid_argument;
// read a given file storing each word in the file in a given vector
void readwords(const string &filename, vector<string> &vec)
{
    // open the file
    ifstream file(filename.c_str());
    // if the open failed, bail out by throwing an exception
    if (!file)
        throw invalid_argument("can't read " + filename);
    // ok: if we get here, the file is open and ready to read
    // first clear vec in case there's any other data there
    vec.clear();
    // next read the file storing each work in the vector we were given
    string s;
    while (file >> s)
        vec.push_back(s);    // add the next word to vec
}
```

Exercises Section 8.4.3

Exercise 8.11: *In the* `open_file` *function, explain why we call* `clear` *before the call to* `open`*. What would happen if we neglected to make this call? What would happen if we called* `clear` *after the* `open`*?*

Answer 8.11: The `ifstream` that is passed to `open_file` might be in a valid or an invalid state. If the `ifstream` is not valid, then any operation will fail. Hence, we must call `clear` before attempting to `open` the stream.

If we neglected to call `clear` and the stream were in some error state, then the call to `open` on the new file would fail, regardless of whether the file named by that `string` is valid.

What happens if we called `clear` after the `open` would depend on the state of `in` when `open_file` was called and on whether the file named by `file` could be opened succsessfully.

If `in` was in an invalid state when `open_file` is called, then the open would fail; `file` would remain attached to whatever file it was bound to when `open_file` was called. The stream would then be cleared. The next IO operation would be against the same file to which `in` was bound *when `open_file` was called*.

If `in` was valid, then the result depends on whether the call to `open` succeeded. If `file` can be successfully opened, then the call to `clear` has no effect—it just resets the stream to the same good state that `in` was in after the `open`. However, if the call to `open` failed, then calling `clear` has to effect of masking the failure of `open`. The stream will appear to be in a good state, however the `open` failed. What happens is undefined, but input operations on `in` are certain to fail in some way.

Exercise 8.14: *Use `open_file` and the program you wrote for the first exercise in Section 8.2 (p. 291) to open a given file and read its contents.*

Answer 8.14:

```
#include <fstream>
#include <string>
#include <vector>
#include <stdexcept>
using std::ifstream; using std::string; using std::vector;
using std::invalid_argument;
ifstream& open_file(ifstream&, const string&);
// read a given file storing each word in the file in a given vector
void readwords(const string &filename, vector<string> &vec)
{
    ifstream file;
    // use open_file, but check return and throw an exception if the open failed
    if (!open_file(file, filename))
        throw invalid_argument("can't read " + filename);
    // ok: if we get here, the file is open and ready to read
    // first clear vec in case there's any other data there
    vec.clear();
    // now read the file storing each work in the vector we were given
    string s;
    while (file >> s)
        vec.push_back(s);    // add the next word to vec
}
```

Exercises Section 8.5

Exercise 8.15: *Use the function you wrote for the first exercise in Section 8.2 (p. 291) to print the contents of an `istringstream` object.*

Answer 8.15:

```
#include <iostream>
#include <sstream>
using std::cin; using std::cout; using std::endl; using std::istringstream;
using std::istream;
istream &readfile(istream &);
```

```
int main()
{
    istringstream str
            ("now is the time for all good boys to come to the aid of the party");
    readfile(str);       // call readfile to read an istringstream
    if (str) {
        cout << "OK: status reset to valid" << endl;
        return 0;    // indicate success
    } else {
        cout << "OOPS: something's wrong, stream is not valid" << endl;
        if (str.eof()) cout << "eof wasn't cleared" << endl;
        if (str.fail()) cout << "fail bit is still set" << endl;
        if (str.bad()) cout << "bad bit is still set" << endl;
        return -1;   // indicate failure
    }
}
```

Exercise 8.16: *Write a program to store each line from a file in a* `vector<string>`*. Now use an* `istringstream` *to read each line from the* `vector` *a word at a time.*

Answer 8.16:

```
#include <fstream>
#include <string>
#include <vector>
#include <stdexcept>
using std::ifstream; using std::string; using std::vector;
using std::invalid_argument;
// read a given file storing each line in the file in a given vector
void readwords(const string &filename, vector<string> &vec)
{
    // open the file
    ifstream file(filename.c_str());
    // if the open failed, bail out by throwing an exception
    if (!file)
        throw invalid_argument("can't read " + filename);
    // ok: if we get here, the file is open and ready to read
    // first clear vec in case there's any other data there
    vec.clear();
    // now read the file storing each work in the vector we were given
    string s;
    while (getline(file, s))
        vec.push_back(s);    // add the next line to vec
}
#include <iostream>
#include <sstream>
using std::cout; using std::endl; using std::cerr; using std::istringstream;
int main()
{
    vector<string> vec;
    try {
        readwords("readwords.cc", vec);
        cout << "OK: here's what was read: " << endl;

        /* print the contents a word at a time, even though vec holds lines
         * we'll bind a istringstream to each element in vec and then
```

```
         * read that istringstream using normal input operator to get the
         * individual words
         */
        for (vector<string>::size_type i = 0; i != vec.size(); ++i) {
            istringstream str(vec[i]);  // bind a stringstream to next element
            string s;                   // s will hold each word in the line
            while (str >> s)            // read and print each word
                cout << "\t" << s << endl;
        }
        return 0;
    } catch (invalid_argument e) {
        cerr << "OOPS: something's wrong: " << e.what() << endl;
        return -1;
    }
}
```

Chapter 9

Exercises Section 9.1.1

Exercise 9.1: *Explain the following initializations. Indicate if any are in error, and if so, why.*

```
int ia[7] = { 0, 1, 1, 2, 3, 5, 8 };
string sa[6] = {
    "Fort Sumter", "Manassas", "Perryville",
    "Vicksburg", "Meridian", "Chancellorsville" };
(a) vector<string> svec(sa, sa+6);
(b) list<int> ilist( ia+4, ia+6);
(c) vector<int> ivec(ia, ia+8);
(d) list<string> slist(sa+6, sa);
```

Answer 9.1:

(a) Initializes `svec` as a copy of the `string` elements in `sa`.
(b) Initializes `ilist` to hold two elements whose initial values are copies of the values `ia[4]` and `ia[5]`.
(c) Error—the initialization incorrectly passes a pointer to `ia + 8`. The array `ia` has 7 elements and so the pointer one past the end of `ia` can be formed as `ia + 7`.
(d) Error—the initializers are swapped. The first initializer represents the beginning of a range of elements and the second represents the iterator one past the end of the range. This call incorrectly passes the end pointer as the first initializer and the beginning pointer as the second.

Exercise 9.3: *Explain the differences between the constructor that takes a container to copy and the constructor that takes two iterators.*

Answer 9.3: To use the constructor that takes a container, the argument and the object being constructed must match exactly—the container and element types must be identical.

We can use the container constructor that takes two iterators to copy elements from an unlike container and/or to copy from a container with a compatible but different element type. For example, we can use the iterator version of the container constructor to copy a `vector<char*>` to a `list<string>` but may not use the constructor that takes a container to construct the `list`. Similarly, we could use the iterator constructor to create a `list<string>` from a `list<char*>`. If we want to use the constructor that takes a container to create a `list<string>`, we may only use a `list<string>` as the initializer.

Exercises Section 9.1.2

Exercise 9.4: *Define a* `list` *that holds elements that are* `deque`s *that hold* `int`s.

Answer 9.4:

```
list< deque<int> > lst;
```

Exercise 9.6: *Given a class type named* `Foo` *that does not define a default constructor but does define a constructor that takes* `int` *values, define a* `list` *of* `Foo` *that holds 10 elements.*

Answer 9.6:

```
list<Foo> lst(10, 0);   // 10 elements each with value of 0
```

Exercises Section 9.2

Exercise 9.7: *What is wrong with the following program? How might you correct it?*

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

Answer 9.7: The `list` iterator does not provide the relational operators such as `<`. We could rewrite the loop to use inequality instead:

```
while (iter1 != iter2) /*   . . .   */
```

Exercise 9.9: *Write a loop to write the elements of a* `list` *in reverse order.*

Answer 9.9: We can write this loop using an iterator. We'll start that iterator at the `end()` of the `list` and decrement it till we reach the `begin()` value. There is one tricky part: handling the fact that the `end()` iterator refers to a nonexistent element one past the end of the container. We handle this fact by decrementing the iterator in the loop before printing the element:

```
list<int> lst;
// give lst some elements
for (int i = 0; i != 10; ++i)
    lst.push_back(i);
// write the elements in reverse order
list<int>::const_iterator it = lst.end();
while (it != lst.begin())
    // remember to decrement it before printing it
    cout << *--it << endl;
```

Exercise 9.10: *Which, if any, of the following iterator uses are in error?*

```
const vector< int > ivec(10);
vector< string >    svec(10);
list< int >         ilist(10);

(a) vector<int>::iterator    it = ivec.begin();
(b) list<int>::iterator      it = ilist.begin()+2;
(c) vector<string>::iterator it = &svec[0];
(d) for (vector<string>::iterator
             it = svec.begin(); it != 0; ++it)
                // ...
```

Answer 9.10: Each of these examples is in error:

(a) `ivec` is a `const vector`, meaning that its elements may not be changed. Attempting to bind a plain `iterator` to `ivec` would allow us to use the iterator to change the elements. We may only bind a `const_iterator` to a `const` container such as `ivec`.
(b) `list` iterators do not support arithmetic, so the usage `ilist.begin()+2` is in error.
(c) The type of the right-hand operand of this example is `string*`—the expression `&svec[0]` fetches the element at position `0` in `svec` and takes the address of that element. The elements in `svec` are `string`s and so the type of `&svec[0]` is a pointer to `string`. There is no conversion from `string*` to `vector<string>::iterator` so the assignment is in error.
(d) The condition in the `for` loop `it != 0` is in error. `it` is a `vector<string>::iterator` and there is no comparison operator from that type to `int`. There also is no automatic conversion from `int` to `vector<string>::iterator`. Thus, `it != 0` is in error.

Exercises Section 9.2.1

Exercise 9.13: *Rewrite the program that finds a value to return an iterator that refers to the element. Be sure your function works correctly if the element does not exist.*

Answer 9.13:

```
list<int>::const_iterator
find_elem(list<int>::const_iterator beg,
          list<int>::const_iterator end, int value)
{
    // keep looking till we find the element or exhaust the input range
    while (beg != end && *beg == value)
        ++beg;
    // beg either refers to the element if present, or to end if not
    return beg;
}
```

Exercise 9.14: *Using iterators, write a program to read a sequence of* `string`*s from the standard input into a* `vector`*. Print the elements in the* `vector`*.*

Answer 9.14:

```
#include <string>
#include <vector>
#include <iostream>
using std::string; using std::vector;
using std::cin; using std::cout; using std::endl;
int main()
{
    vector<string> svec;
    // read strings from the standard input and store them in a vector
    string s;
    while (cin >> s)
        svec.push_back(s);
    // write the elements
    for (vector<string>::const_iterator it = svec.begin();
                                        it != svec.end(); ++it)
        cout << *it << endl;   // print each element
    return 0;
}
```

Exercise 9.15: *Rewrite the program from the previous exercise to use a* `list`*. List the changes you needed to change the container type.*

Answer 9.15: Changing the program to use a `list` instead of a `vector` required only changes to the variable types of `svec` and `it` from `vector` to `list`. Similarly, the `#include` and `using` directives had to be changed to refer to `list` instead of `vector`:

```
#include <string>
#include <list>                // change vector to list
#include <iostream>
using std::string;
using std::list;               // change vector to list
using std::cin; using std::cout; using std::endl;

int main()
{
    list<string> svec;     // change vector to list

    // read strings from the standard input and store them in a vector
    string s;
    while (cin >> s)
        svec.push_back(s);

    // write the elements
    // changed iterator type to vector<string>::iterator
    for (list<string>::const_iterator it = svec.begin();
                                      it != svec.end(); ++it)
        cout << *it << endl;   // print each element

    return 0;
}
```

Exercises Section 9.3.1

Exercise 9.16: *What type should be used as the index into a* `vector` *of* `int`*s?*

Answer 9.16: The type to use as an index into a `vector<int>` is `vector<int>::size_type`.

Exercise 9.17: *What type should be used to read the elments in a* `list` *of* `string`*s?*

Answer 9.17: We should use a `list<string>::const_iterator` to read elements from a `list` of `string`s. If we need to write the `string` elements, then we should use a `list<string>::iterator`.

Exercises Section 9.3.3

Exercise 9.19: *Assuming* `iv` *is a* `vector` *of* `int`*s, what is wrong with the following program? How might you correct the problem(s)?*

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

Answer 9.19: There are several problems with this program:

1. The `while` condition in this loop is nonsensical and will not compile. It defines an `iterator`, which it does not explicitly initialize. The condition then appears to want to compare this uninitialized iterator to `mid`.

2. Even if the condition correctly compared a valid iterator to `mid`, the loop would be in error. The condition compares to the cached iterator value `mid` even though the body inserts elements into the `vector`. Inserting elements anywhere in a `vector` can invalidate iterators and so the iterator used in the `while` condition may become invalid during execution.

3. The `if` statement uses `some_val` in conflicting ways: In the condition it appears that `some_val` is the same type as `iter`, that is a `vector<int>::iterator`. In the body it appers that `some_val` is an `int`.

Fixing the last problem is easiest—we'll assume that `some_val` is an integer and fix the test accordingly:

```
// remember to dereference iter to get a value to compare to some_val
if (*iter == some_val)
    iv.insert(iter, 2 * some_val);
```

Fixing the `while` loop would require understanding what the program wanted to accomplish. One plausible guess as to the programmer's original intention is that the loop should look through the `vector` up to the original midpoint. In the first half of the `vector` the loop should see whether a given value is found. If it is, it should `insert` a new element that is twice the value of the element we were looking for. This element should be inserted immediately ahead of the value we were looking for. Given this specification, we might write the loop as follows:

```
// first remember the index of the original midpoint element
vector<int>::size_type mid_index = iv.size()/2;
// read iv up to but not including the original midpoint
vector<int>::iterator iter = iv.begin();
while (iter != iv.begin() + mid_index) {
    // check whether current element is one we're looking for
    if (*iter == some_val) {
        // if so, insert the new element ahead of iter
        // and reset iter to point to the inserted element
        iter = iv.insert(iter, 2 * some_val);
        // now move iter to refer one past the element equal to some_val
        iter += 2;
        // increment the mid_index because we inserted a new element
        // in the first half of the vector
        ++mid_index;
    } else
        ++iter;   // didn't find a matching element look at the next one
}
```

Exercises Section 9.3.4

Exercise 9.20: *Write a program to compare whether a* `vector<int>` *contains the same elements as a* `list<int>`*.*

Answer 9.20: Because the container equality operators require that the container and element types be identical we cannot use the `==` operator to compare these containers. Instead, we must write the loop explicitly, which we can do as:

```
bool equal = false;
// if the sizes are unequal,then the containers are unequal
if (vec.size() == lst.size()) {
    vector<int>::const_iterator vecit = vec.begin();
    list<int>::const_iterator lstit = lst.begin();
    // we know vec and lst have same number of elements
    // so if vecit is still valid, so is lstit
    while (vecit != vec.end() && *vecit == *lstit) {
        ++vecit;
        ++lstit;
    }
    // the containers are equal if we looked at every element
    equal = vecit == vec.end();
}
```

```
if (equal)
    cout << "ok: they're the same" << endl;
else
    cout << "containers differ" << endl;
```

We start by checking if the two containers are the same size. If not, they can't be equal.

If the two sizes are the same, we iterate through each container comparing elements from the `vector` to the corresponding element in the `list`. The `while` condition checks first that we have not yet exhausted the containers. If we hit the `end()` iterator on `vec` then we've also hit the end on `lst` and the loop is exited.

Assuming the iterators are valid, the second test in the condition checks whether the values to which the iterators refer are equal. If not, we fall out of the loop; if they are equal the body of the `while` increments both iterators so that we look at the next elements in the sequences in the next iteration.

Once the `while` is exited it remains to check whether we exited because the iterators were at the end or because we found an unequal element.

Exercises Section 9.3.5

Exercise 9.23: *What, if any, restrictions does using* `resize` *with a single size argument place on the element types?*

Answer 9.23: Using `resize` requires that the element type be a type that can be value initialized.

When we call `resize`, elements might either be discarded from or added to the container. They are added if the argument to `resize` is greater than the current `size` of the container. If elements are added, they are value initialized: If the container holds a class type, then using `resize` requires that the class have a default constructor. This requirement is the same as the requirement placed on the constructor that takes an element count but no element initializer. It is worth noting that often it is more efficient to use `reserve` rather than `resize`.

Exercises Section 9.3.6

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

Answer 9.24:

```
#include <vector>
#include <iostream>
#include <stdexcept>
using std::vector; using std::cout; using std::endl; using std::out_of_range;
void fetchfirst(const vector<int> &vec)
{
    try {
        cout << vec.at(0) << endl;
    } catch (out_of_range) {
        cout << "no element 0 in vec!" << endl;
    }
    if (vec.size())
        cout << vec[0] << endl;
    else
        cout << "no element 0 in vec!" << endl;
    if (!vec.empty())
        cout << vec.front() << endl;
    else
        cout << "no elements in vec!" << endl;
    if (vec.begin() != vec.end())
        cout << *vec.begin() << endl;
    else
```

```
        cout << "no elements in vec!" << endl;
}
int main()
{
    vector<int> vec;     // empty vector
    fetchfirst(vec);
    vec.push_back(10);   // give vec an element
    fetchfirst(vec);
    return 0;
}
```

Exercises Section 9.3.7

Exercise 9.25: *What happens in the program that erased a range of elements if* `val1` *is equal to* `val2`*. What happens if either* `val1` *or* `val2` *or both are not present.*

Answer 9.25: If `val1` and `val2` are equal then the iterators `elem1` and `elem2` will also be equal. The call to `find` that initializes `elem2` will start its search at `elem1`. That value is equal to `val1`, which in this case would be equal to `val2`. The call to `erase` would attempt to delete the empty range denoted by the equal iterators `elem1` and `elem2`, hence no elements would be removed.

If `val1` is not present then the initial call to `find` will set `elem1` to the end iterator for `slist`. The iterator `elem2` will also be set to that value: Its initializer would call `find` on the empty range denoted by `elem1`—which in this case we know is equal to `slist.end()`—and `slist.end()`. The call to `erase` will be passed an empty range so no elements will be erased.

If `val1` is missing, it doesn't matter whether `val2` is present. The analysis described in the previous paragraph is unchanged regardless of whether `val2` is present.

Exercise 9.27: *Write a program to process a* `list` *of* `strings`*. Look for a particular value and, if found, remove it. Repeat the program using a* `deque`*.*

Answer 9.27:

```
#include <string>
#include <list>
using std::string; using std::list;
void
remelems(list<string> &lst, const string &val)
{
    list<string>::iterator beg = lst.begin();
    while (beg != lst.end())
        if (*beg == val)
            beg = lst.erase(beg);
        else
            ++beg;
}
#include <deque>
using std::deque;
void
remelems(deque<string> &deq, const string &val)
{
    deque<string>::iterator beg = deq.begin();
    // must recalculate the end iterator in case an element is removed
    while (beg != deq.end())
        if (*beg == val)
            beg = deq.erase(beg);
        else
```

```
            ++beg;
}
#include <iostream>
using std::cin; using std::cout; using std::endl;
int main()
{
    list<string> lst;
    deque<string> deq;
    string s;
    while (cin >> s) {
        lst.push_back(s);
        deq.push_back(s);
    }
    cout << "number of elements read: " << deq.size() << endl;
    remelems(lst, "void");
    cout << "number after removing ‘‘void’’" << lst.size() << endl;
    remelems(deq, "void");
    cout << "number after removing ‘‘void’’" << deq.size() << endl;
    return 0;
}
```

Exercises Section 9.3.8

Exercise 9.28: *Write a program to assign the elements from a* `list` *of* `char*` *pointers to C-style character strings to a* `vector` *of* `strings`*.*

Answer 9.28:

```
#include <list>
#include <vector>
#include <string>
using std::string; using std::vector; using std::list;
#include <iostream>
using std::cout; using std::endl;
int main()
{
    list<char*> lst;
    lst.push_back("hello");
    lst.push_back("world");
    vector<string> vec;
    vec.assign(lst.begin(), lst.end());
    for (vector<string>::const_iterator iter = vec.begin();
                                        iter != vec.end(); ++iter)
        cout << *iter << endl;

    return 0;
}
```

Exercises Section 9.4.1

Exercise 9.31: *Can a container have a capacity less than its size? Is a capacity equal to its size desirable? Initially? After an element is inserted? Why or why not?*

Answer 9.31: No, the capacity of a container is greater than or equal to its size. The capacity is the number of elements the container can hold without requiring memory allocation, whereas the size is the number of elements actually in use. Any time we add an element (thus increasing the size) either capacity is greater than the current size plus one or it is equal to the current size. In the latter case, the container is reallocated before the new element is added, thus increasing the capacity before adding to the size of the container.

It is almost always better to define an initially empty container and then add elements than to define a container of a given size. The problem is that when we define a container of a given size, the elements are constructed. The elements are initialized either using the default constructor or a value we supply when creating the container. Unless the elements should all have the same value, this approach is wasteful.

Instead, if we are using a container such as `vector` that has a `reserve` member and we know that the container will grow to at least a given size, it is more efficient to `reserve` the known capacity. In such cases, it is best to define an (initially) empty container and then call `reserve` to allocate the space. Doing so avoids the overhead of construcing elements only to subsequently overwrite them.

Exercise 9.32: *Explain what the following program does:*

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

If the program reads 256 words, what is its likely capacity after it is resized? What if it reads 512? 1,000? 1,048?

Answer 9.32: This program defines an initially empty `vector` to hold `strings` and then allocates enough space to hold 1024 `strings`. Having allocated space, it reads the standard input, storing what was read in the `vector`. If the standard input contains fewer than 1024 `strings`, then the `while` loop does not cause the `vector` to be reallocated. If it reads more then 1024 `strings`, then the `vector` will be reallocated.

Before analyzing the behavior of the `resize` call, it is worth noting that this program, and particularly the call to `resize`, is for illustration purposes. If we really wanted to increase the memory allocated to the `vector` after the `while` loop it would almost surely be more appropriate to use `reserve`, not `resize`.

The call to `resize` might or might not cause the `vector` to be reallocated. It adds half as many (value initialized) elements to the `vector` as were actually read from the standard input. If this new `size` is still less than 1024, the `vector` will not be reallocated.

Specifically, if the program read 256 or 512 words, then the `vector` will not be reallocated by this program. After the `while` loop the `size` would be 256 or 512 respectively and the capacity will be at least 1024. After the `resize` the `size` would be 384 (256 + 256/2) or 768 (512 + 512/2). Both `sizes` are less than the reserved space of 1024, so no allocation should be done.

If the program reads 1000 or 1024 words, then whether the `vector` is reallocated depends on the details of a particular implementation of `vector`.

If the program reads 1000 words, the `vector` will not be reallocated during the `while` loop—the call to `reserve` guaranteed that the capacity would be 1024 or more. The call to `resize` might require a reallocation. We know that the capacity is at least 1024 and the call to `resize` requires a capacity of at least 1500. Depending on the actual capacity allocated when we called `reserve` the `resize` call might or might not require a reallocation.

If the program reads 1048 words, then reallocation might be required inside the `while` and/or by the `resize` operation. Whether and when the reallocation is required depends on how the `vector` implementation manages its capacity.

Exercises Section 9.5

Exercise 9.33: *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 an unknown number of words from a file for the purpose of generating English language sentences.
(b) 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.
(c) Read an unknown number of words. Always insert new words at the back. Remove the next value from the front.
(d) Read an unknown number of integers from a file. Sort the numbers and then print them to standard output.

Answer 9.33:

(a) Until we know how the container will be used to generate sentences, we have no reason to choose one container over another. When there is no obvious reason to prefer a container, the best choice is usually a `vector`, which we would recommend for this task.
(b) As noted, this problem is not well-suited to the sequential containers. We could keep the sorted sequence in a `vector`, which would allow us to use a fast, binary search to locate the position at which to insert the newest element. However, in general that would require that we insert the element in the middle of the `vector`, which can be expensive, particularly if the `vector` is large. Alternatively, we can keep insertion costs low by using a `list`, which would make it inexpensive to insert elements in the middle. However, using a `list` means that we can only use a slow linear search to find where to insert the element. The associative containers, which we cover in the next chapter, are a much better choice for this kind of problem.
(c) This task is ideally suited to using a `deque`, which supports fast insertion or deletion from either end of the container.
(d) This task processes the input sequentially—we can add elements to the end of the container as we read them. Once we've read all the input, sorting the `vector` requires that we move elements around within the container, but there is no reason to add or remove elements from the middle of the `vector`. Thus, a `vector` would be the best choice for this problem.

Exercises Section 9.6

Exercise 9.35: *Use iterators to find and to erase each capital letter from a* `string`*.*

Answer 9.35: We can solve the problem as stated using the following program:

```
#include <string>
#include <iostream>
using std::string; using std::cout; using std::endl;
int main()
{
    string test("AbcDefghIJklMNOP");
    cout << test << endl;
    string::iterator iter = test.begin();
    while (iter != test.end())
        if (isupper(*iter))
            iter = test.erase(iter);
        else
            ++iter;
    cout << test << endl;
    return 0;
}
```

It is worth noting that the performance of this approach will be poor if the `string` is very large. A better solution would not use `erase` which was stipulated as part of the exercise. Instead, we would create a new string and copy into it all the lower case letters from the original:

```
#include <string>
#include <iostream>
using std::string; using std::cout; using std::endl;
int main()
{
    string test("AbcDefghIJklMNOP");
    cout << test << endl;
    string::iterator iter = test.begin();
    // faster way to remove caps is to make a copy of noncap characters
    string result;  // initially empty
    while(iter != test.end()) {
        if (!isupper(*iter))
            result += *iter;
        ++iter;
    }
    cout << result << endl;
}
```

Exercise 9.37: *Given that you want to read a character at a time into a* `string`*, and you know that the data you need to read is at least 100 characters long, how might you improve the performance of your program?*

Answer 9.37: Given that we know a minimum size for a `string` read a character at a time, we could improve performance by reserving enough space to hold the minimum `string` size. In this example, we might call `reserve(100)`.

Exercises Section 9.6.4

Exercise 9.38: *Write a program that, given the* `string`

```
"ab2c3d7R4E6"
```

finds each numeric character and then each alphabetic character. Write two versions of the program. The first should use `find_first_of`*, and the second* `find_first_not_of`*.*

Answer 9.38:

```
#include <string>
#include <iostream>
using std::string; using std::cout; using std::endl;
int main()
{
    string base("ab2c3d7R4E6");
    string nums("0123456789");
    string alpha("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ");
    string::size_type pos = 0;
    while ((pos = base.find_first_of(nums, pos)) != string::npos)
        cout << "number at position " << pos++ << endl;
    pos = 0;  // reset for next test
    while ((pos = base.find_first_of(alpha, pos)) != string::npos)
        cout << "alphabetic character at position " << pos++ << endl;
    pos = 0;  // reset for next test
    while ((pos = base.find_first_not_of(nums, pos)) != string::npos)
        cout << "alphabetic character at position " << pos++ << endl;
    pos = 0;  // reset for next test
```

```
    while ((pos = base.find_first_not_of(alpha, pos)) != string::npos)
        cout << "number at position " << pos++ << endl;
    return 0;
}
```

Exercises Section 9.6.5

Exercise 9.40: *Write a program that accepts the following two* `strings`*:*

```
string q1("When lilacs last in the dooryard bloom'd");
string q2("The child is father of the man");
```

Using the `assign` *and* `append` *operations, create the* `string`

```
string sentence("The child is in the dooryard");
```

Answer 9.40:

```
#include <string>
#include <iostream>
using std::string; using std::cout; using std::endl;
int main()
{
    string q1("When lilacs last in the dooryard bloom'd");
    string q2("The child is father of the man");
    string sentence;  // initially empty
    // copy the initial part of the string from q2
    // the string find operation returns a position, but assign needs an iterator
    // we obtain an iterator by adding the position to the begin iterator
    sentence.assign(q2.begin(), q2.begin() + q2.find("father"));

    // now get iterators denoting the part of the string we want from q1
    string::const_iterator beg = q1.begin() + q1.find("in");
    string::const_iterator end = q1.begin() + q1.find("bloom'd");
    // call append to copy that substring into sentence
    sentence.append(beg, end);
    cout << sentence << endl;
    return 0;
}
```

Exercises Section 9.7.2

Exercise 9.43: *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.*

Answer 9.43:

```
#include <stack>
#include <string>
#include <iostream>
#include <cstddef>
using std::string; using std::stack; using std::size_t;
using std::cin; using std::cout; using std::cerr; using std::endl;
int main()
{
    const char open = '(', close = ')';
```

```
    stack<char> exprs;
    size_t open_cnt = 0;      // number of open parentheses on the stack
    char c;
    // read expression from the standard input
    while (cin >> c) {
        // if we see a close paren, need to pop the parenthesized subexpression
        if (c == close) {
            // bail out if we see a close paren that isn't preceded by an open
            if (!open_cnt) {
                cerr << "unbalanced parentheses!" << endl;
                return -1;
            }
            // pop up to and including the open paren
            while (exprs.top() != open)
                exprs.pop();
            exprs.pop();        // remember to pop the open paren itself
            --open_cnt;         // and to decrement the counter
            exprs.push('X');    // push a value to represent the popped subexpression
        } else {
            exprs.push(c);      // any other character we push onto the stack
            if (c == open)      // remembering to keep track of how many opens
                ++open_cnt;
        }
    }
    // make sure there was a close for every open parenthesis
    if (open_cnt) {
        cerr << "unbalanced parentheses!" << endl;
        return -2;
    }
    // walk the stack, printing each element as it is popped off
    while (!exprs.empty()) {
        cout << exprs.top() << endl;
        exprs.pop();
    }
    return 0;
}
```

Chapter 10

Exercises Section 10.1

Exercise 10.2: *There are at least three ways to create the* `pairs` *in the program for the previous exercise. Write three versions of the program creating the* `pairs` *in each way. Indicate which form you think is easier to write and understand and why.*

Answer 10.2:

```
#include <vector>
#include <utility>
#include <string>
#include <iostream>
using std::string; using std::vector; using std::pair;
using std::cin; using std::cout; using std::endl;
// reads pairs of strings and ints from standard input
void build_vector0(vector< pair<string, int> >& vec)
{
```

```
    string s;
    int i;
    while (cin >> s >> i)
        vec.push_back(pair<string, int>(s, i));
}
// reads pairs of strings and ints from standard input
void build_vector1(vector< pair<string, int> >& vec)
{
    string s;
    int i;
    while (cin >> s >> i)
        vec.push_back(make_pair(s, i));
}
// reads pairs of strings and ints from standard input
void build_vector2(vector< pair<string, int> >& vec)
{
    string s;
    int i;
    while (cin >> s >> i) {
        pair<string, int> p;
        p.first = s;
        p.second = i;
        vec.push_back(p);
    }
}
```

The first and second versions are easiest—they build a temporary `pair` directly in the call to `push_back`. The third way creates a `pair` in which the `first` and `second` members are default initialized. It then overwrites these members with new values without using the initial values.

Which of the first two approaches is easier to write and understand is largely an issue of familiarity. Once programmers are used to using constructors, the first usage may be most natural and direct. Programmers more accustomed to languages with factory methods and/or who are not yet comfortable with C++ constructors might perfer the second form.

Exercises Section 10.2

Exercise 10.4: *Give illustrations on when a* `list`, `vector`, `deque`, `map`, *and* `set` *might be most useful.*

Answer 10.4:

(a) `list` is useful when inserting or deleting from the middle of a sequential data structure. Tree-like data structures are often a good match to `list`. For example, we might use a `list` to keep track of members in a genealogical application that maintains information about a family tree.

(b) `vector` is useful when fast random access is required to the elements of a sequential data structure. Statistical applications are one class of applications that are well-suited to `vectors`. We could read a set of data into a `vector` and analyze a sample of the data by selecting elements randomly.

(c) `deque` is useful when fast random access is required to the elements of a sequential data structure but we must be able to insert and/or delete from both ends of the container. We might use a `deque` in an application in which we wanted to use a `vector` but cannot because adding elements invalidates iterators. A `deque` will not invalidate iterators when elements are added only at the beginning or end of the container, whereas a `vector` may invalidate iterators even if elements are only added at the end.

(d) `map` is useful when we need to be able to fetch a value given an associated key. For example, we might use a `map` as a symbol table in a compiler by storing each identifier as the `map` key and a data structure describing the identifier as the associated value.

(e) `set` is useful when we need to determine quickly whether a container contains a given key. For example, we might use a `set` to keep track of active jobs in an operating system. As the job is started the job ID would be added to the `set` and when a job finished it would be deleted from the `set`.

Exercises Section 10.3.1

Exercise 10.5: *Define a* `map` *that associates words with a* `list` *of line numbers on which the word might occur.*

Answer 10.5:

```
map< string, list<size_t> >
```

Exercise 10.6: *Could we define a* `map` *from* `vector<int>::iterator` *to* `int`*? What about from* `list<int>::iterator` *to* `int`*? What about from* `pair<int, string>` *to* `int`*? In each case, if not, explain why not.*

Answer 10.6:

(a) Yes, assuming the iterators refer to elements in the same `vector`. Although we do not know the precise type represented by `vector<int>::iterator`, we do know that that type must define a < operator. However, we are guaranteed that the < operator works correctly only if the iterators refer to elements in the same container. Because the iterator has a < operator, it can be used as the index in a `map`.
(b) `list<int>::iterator` does not define the relational operators, because the sensible meaning—comparing the relative location of the elements to which the iterators refer—would be unusably expensive. Because the iterator does not define <, we could use this type as a `map` key only by defining an appropriate comparison operation. However, there is no obvious, general meaning for how we might compare two `list<int>::iterators`. In absence of the < operator or some other comparison function, we cannot use the `list iterator` as the key type for a `map`.
(c) Yes, the `pair` type defines the < operator. This operator implements a dictionary ordering on the underlying type using the type's own < operator. In this case, the underlying types, `int` and `string`, support < so we could define a `map` using `pair<int, string>` as the key type.

Exercises Section 10.3.2

Exercise 10.8: *Write an expression using a* `map` *iterator to assign a value to an element.*

Answer 10.8:

```
map<string, int>::iterator iter = /* initialize iter */;
iter->second = 42;   // assigns 42 to the int value of the element to which iter refers
```

Exercises Section 10.3.4

Exercise 10.10: *What does the following program do?*

```
map<int, int> m;
m[0] = 1;
```

Contrast the behavior of the previous program with this one:

```
vector<int> v;
v[0] = 1;
```

Answer 10.10: The first fragment defines an empty `map` and then uses the subscript operator to add an element to the map.

The second fragment is in error: it defines an empty `vector` and then attempts to assign a value to the non-existent first element from the empty `vector`.

Subscripting a `map` and a `vector` have distinctly different behavior: Subscripting a `vector` returns an existing element. Subscripting a `map` fetches the element indexed by the subscript if that element exists. If that index is not already in the `map` then an element with that index is added to the `map`. The value of the inserted element is value-initialized.

Exercise 10.11: *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 could be used to subscript the* `map` *and the type that would be returned from the subscript operator.*

Answer 10.11: Any type that is convertible to the `key_type` of the `map` can be used to index into a `map`. For example, if we had a `map` whose `key` was `int` we could use a value of type `short` or `unsigned` etc. to index the `map`. Given a `map` whose keys are `string`s we could use a C-style character string as the index.

The `map` subscript operator returns a reference to the `mapped_type` of the `map`. The `mapped_type` is the second type named when defining a `map`.

For example, a `map<string, int>` has `key_type` of `const string` and a `mapped_type` of `int`.

Exercises Section 10.3.5

Exercise 10.13: *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.*

Answer 10.13:

```
// the argument type is the map's value_type
typedef pair<const string, vector<int> > arg_type;
// the return type is a pair whose first member is
// the map's iterator and second is a bool
typedef pair< map<string, vector<int> >::iterator, bool > ret_type;
```

Exercises Section 10.3.6

Exercise 10.14: *What is the difference between the* `map` *operations* `count` *and* `find`*?*

Answer 10.14: For `map`, which may contain only one element with a given key, the `count` operation returns zero or one indicating whether the given key is present. The `find` operation for `map` returns an iterator to the element with a given key if the element is present or the `end()` iterator if the key is not found.

Exercise 10.16: *Define and initialize a variable to hold the result of a call to* `find` *on a* `map` *from* `string` *to* `vector` *of* `int`.

Answer 10.16:

```
map< string, vector<int> > m;
// do something that adds elements to m

string some_string;
// do something that gives a value to some_string

map< string, vector<int> >::iterator iter = m.find(some_string);
```

Exercises Section 10.3.9

Exercise 10.17: *Our transformation program uses* `find` *to look for each word:*

```
map<string, string>::const_iterator map_it =
         trans_map.find(word);
```

Why do you suppose the program uses `find`*? What would happen if it used the subscript operator instead?*

Answer 10.17: The program uses `find` because we do *not* want to add an element to `trans_map` if the word we're looking for isn't already there.

If the program used the subscript operator then the effect would be to add every word that is in the input file and not also in the transformation map to the transformation map. Each added word would have a replacement value of the empty string. The effect would be to replace every word that wasn't in the transformation map by the empty string.

Exercise 10.18: *Define a* `map` *for which the key is the family surname and the value is a* `vector` *of the children's names. Populate the* `map` *with at least six entries. Test it by supporting user queries based on a surname, which should list the names of children in that family.*

Answer 10.18:

```
#include <map>
#include <vector>
#include <string>
#include <iostream>
#include <sstream>
#include <fstream>
#include <stdexcept>
using std::map; using std::vector; using std::string;
using std::istringstream; using std::ifstream; using std::range_error;
using std::cin; using std::cout; using std::endl;
/*  reads a file with each family on its own line.
 *  Format is familyname followed by a list of children's names on the same line
 */
void readnames(ifstream &data, map< string, vector<string> > &families)
{
    // read each family name
    string surname;
    // we're starting a new file, so empty anything that's already there
    families.clear();
    while (data >> surname) {
        // now read all the kids names
        string line;                        // get the rest of the line
        getline(data, line);
        istringstream kids_strm(line);      // read the line a name at a time
        string next_child;
        vector<string> kids;                // build a vector to hold the
        while (kids_strm >> next_child)     // kids names
            kids.push_back(next_child);
        families[surname] = kids;           // add this family to the map
    }
}
ifstream& open_file(ifstream&, const string&);
int main(int argc, char *argv[])
{
    // first build the map of surname to children
    // open the specified input file that contains family data
    if (argc < 2)
            throw range_error("No family names file given");
    ifstream families_file;
    if (!open_file(families_file, argv[1]))
        throw range_error("Unable to open family names file");
    // call readnames to populate a map of surname to childrens' names
    map< string, vector<string> > families;
    readnames(families_file, families);
    // now iterate with user to look up kids given the family name
```

```
    string name;
    while (true) {                                      // loop until the user exits
        cout << "Enter a family name or 'q' to quit:" << endl;
        cin >> name;                           // read the name to look for
        if (!cin || name == "q")               // check if the user wants to quit
            break;
        // now see if the user's query name is in the map
        map< string, vector<string> >::iterator iter = families.find(name);

        // if not, inform the user and repeat
        if (iter == families.end())
            cout << name << " not found.  Try again." << endl;
        else {
            // otherwise print each kid's name
            // kids is a reference to the vector associated with name
            vector<string> &kids = iter->second;
            vector<string>::size_type sz = kids.size();   // remember how many kids
            // customize the response to whether 0 or more children
            if (sz == 0)
                cout << name << " has no children";
            else if (sz == 1)
                cout << name << " has one child named " << kids[0];
            else {
                // multiple children print each one followed by a space
                cout << name << " has " << sz << " children named: ";
                for (vector<string>::size_type i = 1; i != sz; ++i)
                    cout << kids[i] << " ";
            }
            cout << endl;   // separate this family from the next
        }
    }
    return 0;
}
```

Exercises Section 10.4.2

Exercise 10.23: *Write a program that stores the excluded words in a* `vector` *instead of in a* `set`*. What are the advantages to using a* `set`*?*

Answer 10.23: Building the `vector` instead of building a `set` is pretty easy: We change the `#include` and type declarations accordingly. We also use the `push_back` member to add elements to the end of the `vector` rather than using `insert` as was done in the `set` version.

The bigger change comes when we try to use the `vector`. In this case, we must manage the lookup ourselves. Chapter 11 will show how we could write this program more efficiently using the standard library algorithms `sort` and `find`. For this implementation we do a linear search, which is probably ok performance-wise given that the exclusion file is small:

```
#include <map>
#include <vector>
#include <string>
#include <fstream>
#include <iostream>
using std::map; using std::vector; using std::string; using std::ifstream;
using std::cin;
void restricted_wc(ifstream &remove_file,
                   map<string, int> &word_count)
{
```

```
    vector<string> excluded;    // vector to hold words we'll ignore
    string remove_word;
    while (remove_file >> remove_word)
        // equivalent to excluded.insert(excluded.end(), remove_word)
        // but push_back is more natural when using a vector
        excluded.push_back(remove_word);
    // read input and keep a count for words that aren't in the exclusion vector
    /* here's where the programs diverge --- to use a vector we'll have to do
     * the search ourselves.  We'll see in the chapter on algorithms a better
     * way to use the vector by using library algorithms.  For now, we'll
     * do a linear search on the assumption that the excluded words file is small
     */
    string word;
    while (cin >> word) {
       // increment counter only if the word is not in excluded
       vector<string>::const_iterator iter = excluded.begin();
       while (iter  != excluded.end() && *iter != word)
           ++iter;
       if (iter == excluded.end())
           ++word_count[word];
    }
}
```

Exercises Section 10.5.2

Exercise 10.27: *Repeat the program from the previous exercise, but this time use* `equal_range` *to get iterators so that you can* `erase` *a range of elements.*

Answer 10.27:

```
#include <map>
#include <string>
#include <iostream>
#include <fstream>
using std::multimap; using std::string;
using std::cin; using std::cout; using std::endl; using std::ifstream;
typedef multimap<string, string> authormap;
typedef authormap::iterator author_iter;
// read file of authors and titles and populate the multimap
// format of the input is assumed to be author last name followed by a title
// NOTE: titles might have embedded spaces
void readbooks(authormap& books, ifstream &data)
{
    string author, title;
    while (data >> author) {
        getline(data, title);
        books.insert(make_pair(author, title));
    }
}
#include <stdexcept>
#include <utility>
using std::range_error; using std::pair;
ifstream &open_file(ifstream&, const string&);
int main(int argc, char **argv)
{
    // read a file to populate authors multimap
    // check that we were given a file to read and can open it successfully
```

```
    if (argc < 2)
        throw range_error("No file name given");
    ifstream in;
    if (!open_file(in, argv[1]))
        throw range_error("Unable to open file");
    // read the file & populate the map
    authormap books;
    readbooks(books, in);
    // now query user for which author to erase
    while (true) {
        string author;
        cout << "Enter author to remove, or 'q' to quit:" << endl;
        cin >> author;
        if (!cin || author == "q")
            break;   // quit when input is exhauated or user says to quit
        // get iterators to range of books for this author
        pair<author_iter, author_iter> iters = books.equal_range(author);
        if (iters.first == iters.second)
            cout << "no entry for " << author << endl;
        else {
            // if there are books, print the titles being removed
            cout << "Erasing books by " << author << ": ";
            while (iters.first != iters.second) {
                cout << iters.first->second << " ";
                ++iters.first;
            }
            // now remove this author from books
            books.erase(author);
            cout << endl;
        }
    }

    return 0;
}
```

Exercise 10.28: *Using the* `multimap` *from the previous exercise, write a program to generate the list of authors whose name begins with the each letter in the alphabet. Your output should look something like:*

```
Author Names Beginning with 'A':
Author, book, book, ...
...
Author Names Beginning with 'B':
...
```

Answer 10.28:

```
#include <map>
#include <string>
#include <iostream>
#include <fstream>
using std::multimap; using std::string;
using std::cout; using std::endl; using std::ifstream;
typedef multimap<string, string> authormap;
typedef authormap::iterator author_iter;
// read file of authors and titles and populate the multimap
// format of the input is assumed to be author last name followed by a title
// NOTE: titles might have embedded spaces
void readbooks(authormap& books, ifstream &data)
```

```
{
    string author, title;
    while (data >> author) {
        getline(data, title);
        books.insert(make_pair(author,title));
    }
}
#include <stdexcept>
using std::range_error;
#include <utility>
using std::pair;
ifstream &open_file(ifstream&, const string&);
/*  Program to print the authors in books in alphabetic order
 *
 *  This program depends upon the fact that in a map all elements
 *  are ordered by key and in a multimap, elements for a given key are
 *  contiguous.  These properties mean that:
 *  (1) we can look for authors by letter using lower_bound.
 *     Either we'll get an element that starts with the given letter
 *     or we'll get the author whose name starts with the letter
 *     nearest in alphabetic order or we'll be off the end of the map.
 *
 *     If lower_bound refers to an element  and is not the end iterator
 *     then we also know that the author field is nonempty.  We know
 *     the author is nonempty because if there are any elements with
 *     the null string for author, those elements will appear at the very
 *     beginning of the map.  The lower_bound for any element that
 *     starts with any letter will be greater than any element that might
 *     have an empty key.
 *
 *  (2) Once we find an author with the letter we want, we can iterate
 *     sequentially through the map until we encounter an author whose
 *     name does not begin with the letter we're processing.
 *
 *  (3) Moreover, the books associated with a given author will appear before
 *     any elements for the next author in alphabetic sequence.  So, we can
 *     print the books sequentially, noting when the author changes to print
 *     the next author's name.
  */
int main(int argc, char **argv)
{
    //  read a file to populate authors multimap
    //  check that we were given a file to read and can open it successfully
    if (argc < 2)
        throw range_error("No file name given");
    ifstream in;
    if (!open_file(in, argv[1]))
        throw range_error("Unable to open file");
    //  read the file & populate the map
    authormap books;
    readbooks(books, in);
    //  now print works by authors alphabetically
    //  We'll use character_set to fetch each character in order
    string character_set("AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoppQqRrSsTtUuVvWwXxYyZz");
    for (string::size_type i = 0; i != character_set.size(); ++i) {
        //  get iterator to first author starting with the letter indexed by i
        string first_letter(1, character_set[i]);
```

```
        author_iter beg = books.lower_bound(first_letter);
        // beg denotes an author starting with the letter indexed by i
        // or books.end() or the author just after where one with the
        // letter indexec by i would be if such an author existed.
        // If we're not at end, we know that the element to which beg
        // refers has a nonempty key, which we can test to see whether
        // the first character in the key field is the letter we want
        if (beg == books.end() || beg->first[0] != character_set[i])
            cout << "no authors starting with letter "
                 << first_letter << endl;
        else {
            // if there are any matching authors print the letter we're processing
            cout << "Authors starting with letter "
                 << first_letter << ": ";
            // so long as the author starts with the letter we want
            // print each matching author and the books associated with that author
            string curr_author;
            while (beg != books.end() && beg->first[0] == character_set[i]) {
                // if there's a new author, reset curr_author and print author name
                if (beg->first != curr_author) {
                    curr_author = beg->first;
                    cout << endl << "\t"
                              << curr_author << " ";
                }
                cout << beg->second << " ";   // print each book title
                ++beg;
            }
            cout << endl;
        }
    }

    return 0;
}
```

Exercises Section 10.6.3

Exercise 10.31: *What is the output of* `main` *if we look for a word that is not found?*

Answer 10.31: The program writes a message that the word we asked for `occurs 0 times`. The relevant part of the program is:

```
// if the word was found, then print count and all occurrences
typedef set<TextQuery::line_no> line_nums;
line_nums::size_type size = locs.size();
cout << "\n" << sought << " occurs "
     << size << " "
     << make_plural(size, "time", "s") << endl;
```

If the word is not found, then the call to `locs.size()` will set `size` to 0.

The body of the `for` loop

```
// print each line in which the word appeared
line_nums::const_iterator it = locs.begin();
for ( ; it != locs.end(); ++it) {
    cout << "\t(line "
          // don't confound user with text lines starting at 0
          << (*it) + 1 << ") "
          << file.text_line(*it) << endl;
}
```

which prints the line numbers will not be executed. On entry to this loop `locs.begin()` will be the same value as returned by `locs.end()`, indicating an empty range. The condition in the `for` will be false on the first iteration and so the loop body will not be executed.

Exercises Section 10.6.4

Exercise 10.33: *Why doesn't the* `TextQuery::text_line` *function check whether its argument is negative?*

Answer 10.33: There is no need to check whether the argument is negative because the argument type is an `unsigned` type. The argument to `text_line` is an `TextQuery::line_no`, which is a typedef for `vector<string>::size_type`. We don't know the preicse type of `vector<string>::size_type` but we are guaranteed that it is an `unsigned` integral type.

Chapter 11

Exercises Section 11.1

Exercise 11.2: *Repeat the previous program, but read values into a* `list` *of* `strings`*.*

Answer 11.2:

```
#include <algorithm>
#include <list>
#include <string>
#include <iostream>
#include <fstream>
#include <stdexcept>
#include <cstddef>
using std::list; using std::count; using std::string;
using std::cin; using std::cout; using std::endl; using std::ifstream;
using std::range_error; using std::size_t;
// declarations for functions we use from elsewhere in the Primer
string make_plural(size_t, const string&, const string&);
ifstream& open_file(ifstream&, const string&);
int main(int argc, char **argv)
{
    // open a file to read into the list of words
    if (argc < 2)
        throw range_error("No file name given");
    ifstream in;
    if (!open_file(in, argv[1]))
        throw range_error("Cannot open specified file");
    // read the file into our list
    list <string> lst;
    string word;
    while (in >> word)
        lst.push_back(word);
    // use count to report how many copies of a given value are given
    string s;
    while (cin >> s) {
        list<string>::size_type cnt = count(lst.begin(), lst.end(), s);
        cout << "\n" << s << " occurs " << cnt
             <<  make_plural(cnt, " time", "s") << endl;
    }
    return 0;
```

```
}
```

Exercises Section 11.2.1

Exercise 11.3: *Use* `accumulate` *to sum the elements in a* `vector<int>`*.*

Answer 11.3:

```
#include <vector>
#include <numeric>
#include <iostream>
using std::vector; using std::accumulate;
using std::cout; using std::endl;
int main()
{
    // define a vector of ints and give the vector some values
    vector<int> vi;
    for (int i = 0; i != 10; ++i)
        vi.push_back(i);
    // now use accumulate to sum the elements
    cout << accumulate(vi.begin(), vi.end(), 0) << endl;
    return 0;
}
```

Exercise 11.4: *Assuming* `v` *is a* `vector<double>` *what, if anything, is wrong with calling* `accumulate(v.begin(), v.end(), 0)`*?*

Answer 11.4: The third argument to `accumulate` in this call has type `int`. That type governs how arithmetic is done inside `accumulate`. The `double` values in `v` will be converted to `int` and added to the `int` parameter using integer addition. When we sum the elements of a `vector` of `double` we presumably would want the addition done using floating point arithmetic. Because the type of the third parameter controls how the summation is done, to arrange for `accumulate` to add the values of `v` as `doubles` we must pass a third parameter whose type is `double`.

Exercises Section 11.2.2

Exercise 11.7: *Determine if there are any errors in the following programs and, if so, correct the error(s):*

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

(b) vector<int> vec;
    vec.reserve(10);
    fill_n(vec.begin(), 10, 0);
```

Answer 11.7:

(a) The program is in error: The `vector vec` is empty so—assuming `lst` is nonempty—the call to `copy` will attempt to write to the nonexistent elements of `vec`. The call to `copy` should use `back_inserter` to allow new elements to be added to `vec`:

```
copy(lst.begin(), lst.end(), back_inserter(vec));
```

would add elements to `vec` to hold copies of the elements in `lst`. After this call, both `lst` and `vec` would hold the same number of elements.

(b) The program is in error: The call to `reserve` allocates space to hold newly allocated elements in `vec` but does not create any elements. We define `vec` as an empty `vector` and after the call to `reserve` `vec` is still empty; it has no elements. Calling `reserve` can affect runtime performance; it leaves the number of elements in the `vector` unchanged.

When we call `fill_n` we attempt to write to the first 10 elements of `vec`, but there are no such elements. Again, we could fix this program by using `back_inserter`:

```
fill_n(back_inserter(vec), 10, 0);
```

Now the call to `fill_n` will add 10 elements to `vec` and give each newly added element the value 0.

Exercise 11.8: *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?*

Answer 11.8: The use of insert iterators such as `back_inserter` can be a slippery concept. Algorithms use iterator operations only, and there are no iterator operations that explicitly change container sizes. The algorithms have no direct access to the underlying container and so cannot perform container operations. Iterator operations (such as `++`, `*`, etc.) let us navigate between elements and let us examine or assign to element values. They do not add or remove elements from the underlying container. Only container operations (such as `insert`, `erase`, etc.) add or remove elements.

When we use `back_inserter` with an algorithm, the algorithm still uses only iterator operations. The algorithm does not directly change the size of the container: What changes the size of the container is `back_inserter`. Insert iterators, such as `back_inserter`, change container sizes implicitly. When an algorithm assigns a value through an insert iterator, the insert iterator adds an element with that value to the container. Hence it is the iterator, not the algorithm, that changes the size of the container.

Exercises Section 11.2.3

Exercise 11.10: *The library defines a* `find_if` *function. Like* `find`*, the* `find_if` *function takes a pair of iterators that indicates a range over which to operate. Like* `count_if`*, it also takes a third parameter that names a predicate that can be used to test each element in the range.* `find_if` *returns an iterator that refers to the first element for which the function returns a nonzero value. It returns its second iterator argument if there is no such element. Use the* `find_if` *function to rewrite the portion of our program that counted how many words are greater than length six.*

Answer 11.10: The portion of the program we need to rewrite is:

```
vector<string>::size_type wc =
             count_if(words.begin(), words.end(), GT6);
```

This solution depends on remembering that the `vector` is sorted by size when we called `count_if`. Because the `vector` is sorted by size, we can call `find_if` to get an iterator denoting the first element of size 6 or greater and subtract that iterator from the `end()` iterator to get the count of elements remaining in `vec`. All of those elements will be of size 6 or greater:

```
vector<string>::iterator it =
             find_if(words.begin(), words.end(), GT6);
vector<string>::size_type wc = words.end() - it;
```

Notice that this program fragment works even if there are no words of size 6 or greater. In that case, the call to `find_if` will return `words.end()`, and the value of `wc` will be `words.end() - words.end()`, which is zero.

Exercise 11.11: *Why do you think the algorithms don't change the size of containers?*

Answer 11.11: The algorithms do not change the size of a container because they do not have access to the container. The algorithms operate on iterators and so can perform only operations that are defined for iterator types. The iterators have no operations that add elements to a container. Instead, they have operations that enable us to read or write an element in a container and to move from one element to another. These actions do not change the size of the container.

Exercises Section 11.3.1

Exercise 11.14: *Write a program that uses* `replace_copy` *to copy a sequence from one container to another, replacing elements with a given value in the first sequence by the specified new value. Write the program to use an* `inserter`, *a* `back_inserter` *and a* `front_inserter`. *Discuss how the output sequence varies in each case.*

Answer 11.14:

```
#include <vector>
#include <list>
#include <iterator>
#include <algorithm>
#include <iostream>
using std::vector; using std::replace_copy; using std::list;
using std::inserter; using std::back_inserter; using std::front_inserter;
using std::cin; using std::cout; using std::endl;

int main()
{
    // first create a container and populate it
    vector<int> vec;
    // give the vector 3 copies of the same range of elements:
    // vec will have 3 elements equal to 0, 3 equal to 1 etc.
    while (vec.size() < 30) {
        for (int i = 0; i != 10; ++i)
            vec.push_back(i);
    }
    // now copy into a list so we can use front_inserter
    list<int> repl, replF, replB;
    replace_copy(vec.begin(), vec.end(), inserter(repl, repl.begin()), 8, 80);
    replace_copy(vec.begin(), vec.end(), front_inserter(replF), 8, 80);
    replace_copy(vec.begin(), vec.end(), back_inserter(replB), 8, 80);

    for (vector<int>::iterator beg = vec.begin(); beg != vec.end(); ++beg)
        cout << *beg << " ";
    cout << endl;

    for (list<int>::iterator beg = repl.begin(); beg != repl.end(); ++beg)
        cout << *beg << " ";
    cout << endl;

    for (list<int>::iterator beg = replF.begin(); beg != replF.end(); ++beg)
        cout << *beg << " ";
    cout << endl;

    for (list<int>::iterator beg = replB.begin(); beg != replB.end(); ++beg)
        cout << *beg << " ";
    cout << endl;

    return 0;
}
```

The sequences created using either `inserter` or `back_inserter` match the original sequence, with the exception that each `8` is replaced by the value `80`. The relative order of the elements is preserved because we copy each element in sequence to the then end of the `list`.

When we use `front_inserter` the order in which elements are inserted is reversed from the order of the original sequence and from the order in which `back_inserter` or `inserter` add elements. Using `front_inserter`, the elements are added in sequence as the then first element in the `list`. When we copy the first element from `vec` into `replF` it is the only element in the `list`. When we copy the second element from `vec` into `replF` that element is inserted in front of the original first element in `replF`. Hence, what is the second element in `vec` becomes the (temporarily) first element in `replF`. As each element is copied from `vec` it is inserted at the beginning of `replF`, revsersing the order of the elements.

Exercise 11.15: *The algorithms library defines a function named* `unique_copy` *that operates like* `unique`*, except that it takes a third iterator denoting a sequence into which to copy the unique elements. Write a program that uses* `unique_copy` *to copy the unique elements from a* `list` *into an initially empty* `vector`*.*

Answer 11.15: Unfortunately, this question has a problem: It relies on knowing material—how to use the `list sort` member—that will be covered later in this chapter. The easiest solution is to rephrase the question (which will be done starting in the fourth printing) to suggest copying unique elements from a `vector` into a `list`.

Given this rewrite to the question, and using the `vector` from the previous exercise, the solution would look something like:

```
// first make sure that vec is in sorted order
sort(vec.begin(), vec.end());
list<int> lst;
unique_copy(vec.begin(), vec.end(), back_inserter(lst));
for (list<int>::iterator beg = lst.begin(); beg != lst.end(); ++beg)
    cout << *beg << " ";
cout << endl;
```

Exercises Section 11.3.2

Exercise 11.16: *Rewrite the program on 410 to use the* `copy` *algorithm to write the contents of a file to the standard output.*

Answer 11.16:

```
#include <iostream>
#include <string>
#include <iterator>
#include <algorithm>
using std::copy; using std::cin; using std::cout;
using std::istream_iterator; using std::ostream_iterator;
using std::string;
int main()
{
    // write one string per line to the standard output
    ostream_iterator<string> out_iter(cout, "\n");
    // read strings from standard input and the end iterator
    istream_iterator<string> in_iter(cin), eof;
    // use copy to read the standard input and write what was read
    // to the standard output, separating each string by a newline
    copy(in_iter, eof, out_iter);
    return 0;
}
```

Exercise 11.18: *Write a program to read a sequence of integer numbers from the standard input using an* `istream_iterator`*. Write the odd numbers into one file, using an* `ostream_iterator`*. Each value should be followed by a space. Write the even numbers into a second file, also using an* `ostream_iterator`*. Each of these values should be placed on a separate line.*

Answer 11.18:

```
#include <iostream>
#include <fstream>
#include <iterator>
#include <algorithm>
using std::copy; using std::cin; using std::cout;
using std::istream_iterator; using std::ostream_iterator;
using std::ofstream;
// takes two input iterators denoting the input sequence and
// two output iterators into which to write the even and odd
// numbers read respectively
void split_evenodd(istream_iterator<int> beg, istream_iterator<int> end,
                   ostream_iterator<int> even, ostream_iterator<int> odd)
{
    // read the input file separating even and odd numbers
    while (beg != end) {
        int i = *beg++;
        if (i % 2)
            *odd++ = i;
        else
            *even++ = i;
    }
}
int main()
{
    // read ints from standard input and the end iterator
    istream_iterator<int> in_iter(cin), eof;
    // write odd numbers to a file using a space as element separator
    ofstream odd("odd");
    ostream_iterator<int> odd_iter(odd, " ");
    // write even numbers to a file using a newline as element separator
    ofstream even("even");
    ostream_iterator<int> even_iter(even, "\n");
    // now call split_evenodd to do the work
    split_evenodd(in_iter, eof, even_iter, odd_iter);
    return 0;
}
```

Exercises Section 11.3.3

Exercise 11.19: *Write a program that uses* `reverse_iterators` *to print the contents of a* `vector` *in reverse order.*

Answer 11.19:

```
#include <vector>
#include <iostream>
using std::vector; using std::cout; using std::endl;
int main()
{
    // define a vector and give it some values
    vector<int> v;
    for (int i = 0; i != 10; ++i)
        v.push_back(i);
    // now use reverse iterators to print it backwards
    vector<int>::reverse_iterator beg = v.rbegin();
    while (beg != v.rend())
        cout << *beg++ << endl;
```

```
    return 0;
}
```

Exercise 11.20: *Now print the elements in reverse order using ordinary iterators.*

Answer 11.20:

```
// use ordinary forward iterators and -- to print elements in reverse order
vector<int>::iterator end = v.end();
while (end != v.begin())
    cout << *--end << endl;   // remember to decrement before printing!
```

Exercises Section 11.3.5

Exercise 11.25: *What kinds of iterators do you think* `copy` *requires? What about* `reverse` *or* `unique`*?*

Answer 11.25: To infer what kinds of iterators are needed, we should list the operations likely to be needed for each iterator argument.

The `copy` algorithm takes three iterators. The first two denote an input range whose elements are read in sequence. The third iterator refers to a destination sequence into which elements are written in sequence. There is no need to be able to write to the input sequence or to read from the output sequence. These requirements suggest that the first two iterators are input iterators and the third is an output iterator.

We haven't used the `reverse` algorithm as yet, but can infer from its name that it reverses the elements of a sequence. To reverse the elements of a sequence, we would need to read and write the elements. Presumably a `reverse` operation would swap elements starting at both ends of the container, swapping `*begin()` with `*end() - 1`, then swapping `(*begin() + 1` with `*end() - 2` etc. until the entire range is swapped. Such an algorithm would need both `++` and `--` on the iterator. Hence, we can infer that `reverse` would require a bidirectional iterator.

Having used `unique` we know that it takes a pair of iterators denoting an input range. We also know that the algorithm writes its output back into the same container that it reads. Hence, we know that `unique` needs to read and write its elements, implying that the algorithm requires at least a forward iterator.

Exercise 11.26: *Explain why each of the following is incorrect. Identify which errors should be caught during compilation.*

```
(a) string sa[10];
    const vector<string> file_names(sa, sa+6);
    vector<string>::iterator it = file_names.begin()+2;

(b) const vector<int> ivec;
    fill(ivec.begin(), ivec.end(), ival);

(c) sort(ivec.begin(), ivec.rend());

(d) sort(ivec1.begin(), ivec2.end());
```

Answer 11.26:

(a) `file_names` is a `const vector` so we cannot obtain a plain iterator into the `vector`. The declaration of `it` should be:

```
vector<string>::const_iterator it = file_names.begin()+2;
```

(b) The call to `fill` should fail to compile because the iterators returned by `ivec.begin` and `ivec.end` will be `const_iterators`. The `vector ivec` is a `const vector` and so when we call `begin` or `end` the type of iterator we get is a `const_iterator`. Inside `fill`, the code must attempt to assign to the iterator it is passed. We can infer that there is a loop something like:

```
// psuedo-code implementation of fill
while (beg != end)
    *beg++ = val;
```

where `beg` and `end` are `fill`'s iterator parameters and `val` is third `fill` parameter. However, in this call, the iterators passed to `fill` are `const_iterators` and it is not possible to assign to a `const_iterator`.

(c) This call attempts to use a plain iterator and a `reverse_iterator` to denote an input range. Instead, we can sort in ascending order using `begin()` and `end()` or in descending order using `rbegin()` and `rend()`:

```
sort(ivec.begin(), ivec.end());     // ascending order
sort(ivec.rbegin(), ivec.rend());   // descending order
```

(d) This call is in error because it attempts to use iterators referring to elements in two different containers.

Exercises Section 11.4.2

Exercise 11.27: *The library defines the following algorithms:*

```
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);
```

Based only on the names and parameters to these functions, describe the operation that these algorithms perform.

Answer 11.27: In all four functions we can infer that:

- `beg` and `end` are iterators that denote an input range;
- `dest` is an output iterator that denotes a destination container;
- `old_val` is the value to be replaced;
- `pred` is a predicate function used to determine which elements are replaced.
- `old_val` is the new value to use in place of the one being replaced.

Given this interpretation of the parameters, we can expect that

`replace` replaces elements in the input range denoted by `beg` and `end` that are equal to `old_val` by the new value `new_val`.

`replace_if` replaces elements in the input range denoted by `beg` and `end` for which `pred` returns true by the new value `new_val`.

`replace_copy` replaces elements in the input range denoted by `beg` and `end` that are equal to `old_val` by the new value `new_val` and writes the new sequence into `dest` leaving the original sequence unchanged.

`replace_copy_if` replaces elements in the input range denoted by `beg` and `end` for which `pred` returns true by the new value `new_val` and writes the new sequence into `dest` leaving the original sequence unchanged.

Exercises Section 11.5

Exercise 11.29: *Reimplement the program that eliminated duplicate words that we wrote in Section 11.2.3 (p. 400) to use a* `list` *instead of a* `vector`*.*

Answer 11.29: As usual when switching from one type of container to another, the type declarations in the program must be changed. In this case, we change the program to use `list` rather than `vector`. The other change we must make is to use the `list`-specific `sort` member rather than calling the generic `sort` algorithm.

```
#include <list>
#include <string>
#include <cstddef>
#include <iostream>
#include <fstream>
using std::list; using std::string; using std::size_t;
using std::cin; using std::cout; using std::endl; using std::ifstream;
int main()
{
    list<string> words;     // use a list instead of a vector
    // copy contents of each book into a single list
    string next_word;
    while (cin >> next_word) {
        // insert next book's contents at end of words
        words.push_back(next_word);
    }
    // sort words alphabetically so we can find the duplicates
    words.sort();           // use the list member not the library algorithm
    /* eliminate duplicate words:
     * unique reorders words so that each word appears once in the
     *      front portion of words and returns an iterator
     *      one past the unique range;
     * erase uses that iterator as the beginning of the range
     *      to erase, after erase only the unique words remain
     */
    // the remaining code changes only to use list in type declarations;
    // otherwise the code is unchanged from the vector version in the book
    list<string>::iterator end_unique =
                unique(words.begin(), words.end());
    words.erase(end_unique, words.end());

    // print contents to see that duplicates are gone
    for(list<string>::iterator it = words.begin(); it != words.end(); ++it)
        cout << *it << " ";
    cout << endl;
    return 0;
}
```

Chapter 12

Exercises Section 12.1.1

Exercise 12.4: *Indicate which members of* `Person` *you would declare as* `public` *and which you would declare as* `private`*. Explain your choice.*

Answer 12.4:

```
#include <string>
// class Person represents the name and address of a person
class Person {
public:
```

```
    // constructor and access functions are public

    // constructor that takes two strings used to initialize member data
    Person(const std::string &n, const std::string &a):
                                               nm(n), addr(a) { }
    // functions to return name and address;
    // these functions are const because they do not change the data members
    std::string name() const { return nm; }
    std::string address() const { return addr; }
private:
    // class with two string members to represent name and address
    // data members are private to prevent user access to the representation
    std::string nm;
    std::string addr;
};
```

Exercises Section 12.1.2

Exercise 12.6: *How do classes defined with the* `class` *keyword differ from those defined as* `struct`*?*

Answer 12.6: Classes defined using `class` and `struct` differ as to which access label is used for members defined before the first explicit access label inside the class definition. In a class defined using the `class` keyword the access of members prior to the first explicit label is `private`; in classes defined using `struct` those members are `public`.

Exercises Section 12.1.3

Exercise 12.10: *Explain each member in the following class:*

```
class Record {
    typedef std::size_t size;
    Record(): byte_count(0) { }
    Record(size s): byte_count(s) { }
    Record(std::string s): name(s), byte_count(0) { }
    size byte_count;
    std::string name;
public:
    size get_count() const { return byte_count; }
    std::string get_name() const { return name; }
};
```

Answer 12.10: It is worth noting that the constructors and typedef in this class are private. Ordinarily, constructors should be made public and the typedef is used in the interface of public members indicating that it also should be public. The fact that the constructors and typedef are private is an error in the book, which will be corrected in the fourth and subsequent printings.

`size` is defined as type name that is a synonym for `std::size_t`.

The next three members are constructors.

- The first constructor takes no arguments, which means that it is the default constructor. It explicitly initializes the `byte_count` member to zero and implicitly initializes `name` to the empty string.
- The second takes an argument of type `Record::size` (which is a typedef that defines a synonym for the type `size_t`). This constructor explicitly initializes the `byte_count` member to the argument's value. It implicitly initializes `name` to the empty string.
- The final constructor takes a single `string` argument, which it uses to initialize the `name` member. This constructor also explicitly initializes `byte_count` to zero.

- The next two members, `byte_count` and `name`, are the data members of `Record`. The `byte_count` member uses the `Record::size` typedef to define a member of type `size_t` and the `name` member is a `string`.
- The `public` members `get_count` and `get_name` provide access to the `private` data members of the class. Both functions return the related data member as a value, not by reference: The user can read but not write to the class' data. Because the data cannot be written, these members are `const`. Had the functions returned (non`const`) references to the data members, then these functions could not be `const`.

Exercises Section 12.1.4

Exercise 12.11: *Define a pair of classes* `X` *and* `Y`*, in which* `X` *has a pointer to* `Y`*, and* `Y` *has an object of type* `X`*.*

Answer 12.11:

```
class Y;      // forward declaration for pointer member in X
// must define X before defining Y because Y has a member of type X
class X {
     Y* ptr;          // ok: pointer to an incomplete type
     // other members of X
};
class Y {
     X member;       // ok: class X is defined
     // other members of Y
};
```

Exercises Section 12.2

Exercise 12.13: *Extend your version of the* `Screen` *class to include the* `move`*,* `set`*, and* `display` *operations. Test your class by executing the expression:*

```
// move cursor to given position, set that character and display the screen
myScreen.move(4,0).set('#').display(cout);
```

Answer 12.13:

```
#include <string>
class Screen {
public:
     typedef std::string::size_type index;
     // interface member functions
     // constructor: build screen of given size containing all blanks
     Screen(index ht = 0, index wd = 0):
                  contents(ht * wd, ' '), cursor(0),
                  height(ht), width(wd) { }
     // new members to move the cursor, set a character in the screen,
     // and display the screen contents
     Screen& move(index r, index c);
     Screen& set(char);
     Screen& set(index, index, char);
     // display overloaded on whether the object is const. Returns
     // a plain or const reference depending on whether the object is const
     Screen& display(std::ostream &os)
                  { do_display(os); return *this; }
     const Screen& display(std::ostream &os) const
                  { do_display(os); return *this; }
     // return character at the cursor or at a given position
```

```
        char get() const { return contents[cursor]; }
        char get(index ht, index wd) const;
        // remaining members
    private:
        // single function to do the work of displaying a Screen,
        // will be called by the display operations
        void do_display(std::ostream &os) const
                          { os << contents; }
        std::string contents;
        index cursor;
        index height, width;
    };
    Screen& Screen::set(char c)
    {
        contents[cursor] = c;
        return *this;
    }
    Screen& Screen::move(index r, index c)
    {
        index row = r * width; // row location
        cursor = row + c;
        return *this;
    }
    Screen& Screen::set(index r, index c, char ch)
    {
        index row = r * width; // row location
        contents[row + c] = ch;
        return *this;
    }
    char Screen::get(index r, index c) const
    {
        index row = r * width; // row location
        return contents[row + c];
    }
```

Exercises Section 12.3

Exercise 12.16: *What would happen if we defined* `get_cursor` *as follows:*

```
index Screen::get_cursor() const
{
    return cursor;
}
```

Answer 12.16: The code would fail to compile—the return type uses a typedef defined inside the `Screen` class. But when we define a member function outside the class header, the class members are not in scope until the class name is seen. Thus, to use the `index` member as the return type, we must explicitly qualify the name `index` to indicate that we want the one from the `Screen` class.

Exercises Section 12.3.1

Exercise 12.18: *Explain the following code. Indicate which definition of* `Type` *or* `initVal` *is used for each use of those names. If there are any errors, say how you would fix the program.*

```
typedef string Type;
Type initVal();

class Exercise {
public:
    // ...
    typedef double Type;
    Type setVal(Type);
    Type initVal();
private:
    int val;
};

Type Exercise::setVal(Type parm) {
    val = parm + initVal();
}
```

The definition of the member function `setVal` *is in error. Apply the necessary changes so that the class* `Exercise` *uses the global typedef* `Type` *and the global function* `initVal`.

Answer 12.18: This code starts by defining a global typedef named `Type` as a synonym for `string`. The declaration `initVal` uses this typedef, meaning that the function returns a `string`. The class `Exercise` defines its own member named `Type` that is also a type name. Inside `Exercise`, the name `Type` is a synonym for `double`. The `Exercise` members, `setVal` and `initVal` are defined after the definition of `Type`. Their return types, therefore, refer to the name `Type` as defined inside the `Exercise` class. These functions return `double`.

The definition of `Exercise::setVal` is in error because the return type in the definition does not match the return type used in the declaration of the function inside `Exercise`. The definition uses the global typedef, whereas the declaration used the definition of `Type` that is a member of the `Exercise` class.

```
typedef string Type;
Type initVal();          // uses the global definition of Type; initVal returns a string
class Exercise {
public:
    // ...
    typedef double Type;    // hides the global definition of Type
    ::Type setVal(::Type); // uses global definition of Type
    ::Type initVal();       // uses global definition of Type
private:
    int val;
};
// Changed so that the parameter and return type refer to the global definition of Type.
::Type Exercise::setVal(::Type parm) {
    val = parm + ::initVal();    // calls global definition of initVal
}
```

Exercises Section 12.4

Exercise 12.19: *Provide one or more constructors that allows the user of this class to specify initial values for none or all of the data elements of this class:*

```
class NoName {
public:
    // constructor(s) go here ...
private:
    std::string  *pstring;
    int          ival;
    double       dval;
};
```

Explain how you decided how many constructors were needed and what parameters they should take.

Answer 12.19: The class as presented gives no obvious guidance on which member values the user should be allowed to provide and which should be set by default. Hence, we define the entire set of combinations, allowing the user to selectively define zero or more of the members:

```
#include <string>
class NoName {
public:
    // constructors: all data members are built-in types and must be explicitly initialized
    // default constructor, takes no arguments
    NoName(): pstring(0), ival(0), dval(0) { }
    // constructor to initialize the pointer member
    NoName(std::string *p): pstring(p), ival(0), dval(0) { }
    // constructor to initailize the int member
    NoName(int i): pstring(0), ival(i), dval(0) { }
    // constructor to initialize the double member
    NoName(double d): pstring(0), ival(0), dval(d) { }
    // constructor to initialize the int and double members
    NoName(int i, double d): pstring(0), ival(i), dval(d) { }
    // constructor to initialize the pointer and int members
    NoName(std::string *p, int i): pstring(p), ival(i), dval(0) { }
    // constructor to initialize the pointer and double members
    NoName(std::string *p, double d): pstring(p), ival(0), dval(d) { }
    // constructor to initialize all three members
    NoName(std::string *p, int i, double d): pstring(p), ival(i), dval(d) { }
private:
    std::string  *pstring;
    int          ival;
    double       dval;
};
```

Exercises Section 12.4.1

Exercise 12.22: *The following initializer is in error. Identify and fix the problem.*

```
struct X {
    X (int i, int j): base(i), rem(base % j) { }
    int rem, base;
};
```

Answer 12.22: The initializer for `rem` is in error because it uses the value of the member `base` before `base` has been initialized. Data members are initialized in the order in which they are defined inside the class.

Because `rem` is defined first, it is initialized before `base`.

We could fix this code by reversing the order of the class data members so that `base` preceeds `rem`. However, it is safer to define initializers so that they do not refer to the other data members of the class, which we could do as follows:

```
X (int i, int j): rem(i % j), base(i) { }
```

Now the order in which the data members are defined no longer matters. Note that we have also reordered the member initializers so that they conform to the order in which the initializers will be applied.

Exercise 12.23: *Assume we have a class named* `NoDefault` *that has a constructor that takes an* `int` *but no default constructor. Define a class* `C` *that has a member of type* `NoDefault`*. Define the default constructor for* `C`*.*

Answer 12.23:

```
#include <string>
class NoDefault {
public:
    NoDefault(int);
    // additional members follow, but no other constructors
};
class C {
    NoDefault ND_member;       // NoDefault requires an int initializer
    std::string illustration; // for illustration purposes
    // other members of C
public:
    C(): ND_member(0) { }      // we must explicitly initialize ND_member
                               // ok to implicitly initialize illustration
    // remaining members of C
};
```

Exercises Section 12.4.2

Exercise 12.24: *Using the version of* `Sales_item` *from page 458 that defined two constructors, one of which has a default argument for its single* `string` *parameter, determine which constructor is used to initialize each of the following variables and list the values of the data members in each object:*

```
Sales_item first_item(cin);

int main() {
    Sales_item next;
    Sales_item last("9-999-99999-9");
}
```

Answer 12.24: The object `first_item` is initialized using the constructor that takes a reference to an `istream`. The object's members are initialized using whatever values are supplied on the standard input.

The local object `next` is initialized using the default constructor. That constructor takes an optional `string` parameter. In this call, we use the default empty string argument. The resulting object has an `isbn` member that is the null string and `revenue` and `unit_sold` members that are zero.

The local object `last` is initialized using the constructor that takes a `string` but does not use the default argument. The user supplied argument, `"9-999-99999-9"` is converted to a `string` and that value is used to initialize the `isbn` member. The `revenue` and `unit_sold` members are zero.

Exercise 12.26: *Would it be legal for both the constructor that takes a* `string` *and the one that takes an* `istream&` *to have default arguments? If not, why not?*

Answer 12.26: We could define both constructors with a default argument, but would be unable to use either constructor unless we actually supply arguments. The problem is that it would be impossible to determine which constructor to call when the user supplies no initializer.

Exercises Section 12.4.3

Exercise 12.27: *Which, if any, of the following statements are untrue? Why?*

(a) A class must provide at least one constructor.
(b) A default constructor is a constructor with no parameters for its parameter list.
(c) If there are no meaningful default values for a class, the class should not provide a default constructor.
(d) If a class does not define a default constructor, the compiler generates one automatically, initializing each data member to the default value of its associated type.

Answer 12.27:

(a) False—a class is not required to define any constructors.
(b) False—The default constructor must be callable with no arguments; the constructor may have multiple parameters so long as they each have a default argument.
(c) Depends—Classes that do not define a default constructor have limitations that can be problematic. It is usually a good idea to define a default constructor even when nonobvious initial values must be invented to use to initialize the data members. However, in some cases the difficulty of defining appropriate defaults can outweigh the limitations of not having a default constructor. So, although most classes ordinarily ought to have a default constructor, it is possible for perfectly good reasons to define a class that does not have a default constructor.
(d) False—The compiler synthesizes the default constructor only if a class defines no other constructors. Also, the synthesized constructor uses the same rules as variable initialization: Members of class type are guaranteed to be initialized by running the member's own default constructor. Members of built-in type are initialized only for non-static objects. The built-in members of local nonstatic objects of the class type are uninitialized.

Exercises Section 12.4.4

Exercise 12.29: *Explain what operations happen during the following definitions:*

```
string null_isbn = "9-999-99999-9";
Sales_item null1(null_isbn);
Sales_item null("9-999-99999-9");
```

Answer 12.29:

`null_isbn` is initialized by first constructing a temporary using the `string` constructor that takes a `const char*`. That temporary is then copied into `null_isbn` using the `string` constructor that takes a reference to a `const string`.

`null1` is constructed using the `Sales_item` constructor that takes a `string`.

`null` is constructed by first using the `string` constructor that takes a `const char*` to create a temporary `string` and then passing that temporary to the `Sales_item` constructor that takes a `string`.

Exercises Section 12.4.5

Exercise 12.31: *The data members of* `pair` *are* `public`*, yet this code doesn't compile. Why?*

```
pair<int, int> p2 = {0, 42};   // doesn't compile, why?
```

Answer 12.31: The code fails to compile because the `pair` class defines a constructor. To use explicit member initialization, all members must be public and the class must not define any constructors.

Exercises Section 12.5

Exercise 12.32: *What is a friend function? A friend class?*

Answer 12.32: A friend function is a function named as a friend by a given class. Unlike nonfriend functions, a friend function may access the nonpublic members of the class that grants it friendship.

Similarly, a class may designate another class as its friend. The member functions of the friend class may access the nonpublic members of the class that grants it friendship. The class granting friendship has no reciprocal access to the friend class's nonpublic members.

Exercise 12.35: *Define a nonmember function that reads an* `istream` *and stores what it reads into a* `Sales_item`*.*

Answer 12.35:

```
ifstream &read(ifstream &in, Sales_item &item)
{
    // read isbn and number of copies for a transaction
    in >> item.isbn >> item.units_sold;

    // get the price per copy and calculate the total revenue
    double price;
    in >> price;

    // make sure the read succeeded before using price
    if (in)
        item.revenue = item.units_sold * price;
    else {
        // if the read failed, reset item to indicate something's wrong
        item.revenue = 0;
        item.units_sold = 0;
        item.isbn = string();
    }
    return in;
}
```

Because this function uses members of its `Sales_item` parameter, `item`, the `read` function must be declared as a friend by `Sales_item`:

```
class Sales_item {
    friend std::ifstream &read(std::ifstream &in, Sales_item &item);
    // as before
};
```

Exercises Section 12.6.1

Exercise 12.40: *Using the classes from the previous two exercises, add a pair of* `static` *member functions to class* `Bar`*. The first* `static`*, named* `FooVal`*, should return the value of class* `Bar`*'s* `static` *member of type* `Foo`*. The second member, named* `callsFooVal`*, should keep a count of how many times* `xval` *is called.*

Answer 12.40: Note: There is a typo in this question that will be fixed in the fourth and subsequent printings. The last sentence reference to `xval` should have been to `FooVal`.

The solution given this correction to the problem is:

```
class Foo {
    int member;
public:
    Foo(int i): member(i) { }
    int get_mem() const { return member; }
```

```
};

class Bar {
    static int int_mem;
    static Foo foo_mem;
public:
    static int FooVal() { callsFooVal(); return foo_mem.get_mem(); }
    static void callsFooVal() { ++int_mem; }
    static int calls_cnt() { return int_mem; }
};
```

Exercises Section 12.6.2

Exercise 12.41: *Given the classes* `Foo` *and* `Bar` *that you wrote for the exercises to Section 12.6.1 (p. 470), initialize the* `static` *members of* `Foo`*. Initialize the* `int` *member to 20 and the* `Foo` *member to 0.*

Answer 12.41: Again, there is a typo in this question, the static members to initialize are the members of `Bar`, not `Foo`.

```
// define and initialize the static members of Bar
Foo Bar::foo_mem(0);
int Bar::int_mem = 20;
```

Exercise 12.42: *Which, if any, of the following* `static` *data member declarations and definitions are errors? Explain why.*

```
// example.h
class Example {
public:
    static double rate = 6.5;
    static const int vecSize = 20;
    static vector<double> vec(vecSize);
};

// example.C
#include "example.h"
double Example::rate;
vector<double> Example::vec;
```

Answer 12.42:

- The declaration of `rate` is in error because only `const` integral static members may be initialized in the class body.

- The declaration of `vecSize` is fine. This member is a `static` that has a `const` integral type, which means that it can be initialized inside the class body.

- The declaration of `vec` is in error: The declaration supplies an initializer, `vecSize`, but `vec` is a `vector`. We can only initialize `const` integral static members inside the class body.

- The definitions of both `rate` and `vec` are ok. The `rate` member is value initialized, meaning that it is set to zero. The `vec` member is initialized using the default `vector` constructor meaning that it is an initially empty `vector`.

Chapter 13

Exercises Section 13.1

Exercise 13.2: *The second initialization below fails to compile. What can we infer about the definition of* `vector`*?*

```
vector<int> v1(42);   // ok: 42 elements, each 0
vector<int> v2 = 42;  // error: what does this error tell us about vector?
```

Answer 13.2: We can infer that the `vector` constructor that takes an `int` is an explicit constructor. The initialization of `v1` uses this constructor directly, which is fine. The initialization of `v2` implicitly uses this constructor to create a temporary `vector` and then uses the copy constructor to initialize `v2` from that temporary. Given that the constructor that takes an `int` is explicit, using the constructor implicitly in this way would be illegal.

Exercise 13.3: *Assuming* `Point` *is a class type with a* `public` *copy constructor, identify each use of the copy constructor in this program fragment:*

```
Point global;

Point foo_bar(Point arg)
{
    Point local = arg;
    Point *heap = new Point(global);
    *heap = local;
    Point pa[ 4 ] = { local, *heap };
    return *heap;
}
```

Answer 13.3: The copy constructor is used:

1. To initialize the `arg` parameter from the argument passed in a call to `foo_bar`.
2. To initialize `local` as a copy of `arg`.
3. To initialize the dynamically allocated `Point` addressed by `heap`.
4. To initialize the first two elements in the array `pa`.
5. To initialize the return value from `*heap`.

Exercises Section 13.1.2

Exercise 13.5: *Which class definition is likely to need a copy constructor?*

(a) A `Point3w` *class containing four float members*
(b) A `Matrix` *class in which the actual matrix is allocated dynamically within the constructor and is deleted within its destructor*
(c) A `Payroll` *class in which each object is provided with a unique ID*
(d) A `Word` *class containing a* `string` *and a* `vector` *of line and column location pairs*

Answer 13.5:

(a) The synthesized copy constructor is likely to work fine for the `Point3w` class.
(b) The `Matrix` class needs a copy constructor to make a copy of the dynamically allocated storage. Because the destructor destroys the matrix, each `Matrix` object needs its own copy.
(c) Whether the `Payroll` class needs a copy constructor depends on the detailed design of this class. It is possible that we should not be allowed to copy `Payroll` objects. Alternatively, it is possible that copying a `Payroll` should copy all the corresponding information. That is, copying an object should

not generate a new ID, but should copy the existing ID. In the first case, we would make the copy constructor private in order to prevent copies. In the second, we could use the default copy constructor, which would copy the ID.

(d) The `Word` class probably can use the synthesized constructor. That constructor will delegate to the `string` and `vector` classes the job of copying the member data.

Exercise 13.6: *The parameter of the copy constructor does not strictly need to be* `const`*, but it does need to be a reference. Explain the rationale for this restriction. For example, explain why the following definition could not work.*

```
Sales_item::Sales_item(const Sales_item rhs);
```

Answer 13.6: The parameter must be a reference because otherwise there would be no way to initialize the parameter. A nonreference parameter is initialized as a *copy* of its corresponding argument. The compiler uses the copy constructor to make a copy of a class type object. If the parameter were a nonreference type, then the copy constructor would have to call the copy constructor to initialize its parameter. But that copy constructor would need to call the copy constructor to initialize its parameter, and so on in an infinite recursion.

Exercises Section 13.2

Exercise 13.8: *For each type listed in the first exercise in Section 13.1.2 (p. 481) indicate whether the class would need an assignment operator.*

Answer 13.8:

(a) The synthesized assignment operator is likely to work fine for the `Point3w` class.
(b) The `Matrix` class needs an assignment operator to destroy the matrix in the left-hand operand and allocate a new matrix as a copy of the dynamically allocated storage from the right-hand.
(c) Whether the `Payroll` class needs an assignment operator depends on the details of what this class represents. As with the copy constructor, it is possible that assignment should made `private` to prevent users from assigning objects of this type. Alternatively, we might use the default assignment operator to copy all the data *including* the ID from the right-hand operand into this object.
(d) The `Word` class probably can use the synthesized assignment operator. That operator will delegate to the `string` and `vector` classes the job of assigning the member data.

Exercise 13.10: *Define an* `Employee` *class that contains the employee's name and a unique employee identifier. Give the class a default constructor and a constructor that takes a* `string` *representing the employee's name. If the class needs a copy constructor or assignment operator, implement those functions as well.*

Answer 13.10:

```
#include <string>
class Employee {
    std::string name;
    std::string ID;
    static std::string next_ID();   // generate a new unique ID
public:
    // default constructor implicitly initializes name and generates a new ID
    Employee(): ID(next_ID()) { }
    Employee(const std::string &n): name(n), ID(next_ID()) { }
private:
    // copy constructor and assignment operator are private to prevent copies
    Employee(const Employee&);
    Employee& operator=(const Employee&);
};
```

Exercises Section 13.3

Exercise 13.12: *Determine whether the* `NoName` *class skteched in the exercises on page 481, is likely to need a destructor. If so, implement it.*

Answer 13.12: This class will need a destructor to destroy the `string` pointed to by the `pstring` member. It might look something like:

```
NoName::~NoName() { delete pstring; }
```

Exercise 13.13: *Determine whether the* `Employee` *class, defined in the exercises on page 484, needs a destructor. If so, implement it.*

Answer 13.13: The `Employee` class (as defined so far) has no need for a destructor.

Exercise 13.15: *How many destructor calls occur in the following code fragment?*

```
void fcn(const Sales_item *trans, Sales_item accum)
{
    Sales_item item1(*trans), item2(accum);
    if (!item1.same_isbn(item2)) return;
    if (item1.avg_price() <= 99) return;
    else if (item2.avg_price() <= 99) return;
    // ...
}
```

Answer 13.15: In the code shown there are four exit points from this function: the three `return` statements and the implicit return at the end of the function. Destructors for local objects are called at every exit point.

There are three local `Sales_item` objects: The parameter named `accum` and the two locally defined objects, `item1` and `item2`. On any given execution of the function, three destructors will be run.

Exercises Section 13.4

Exercise 13.18: *Write the corresponding* `Folder` *class. That class should hold a* `set<Message*>` *that contains elements that point to* `Messages`*.*

Answer 13.18:

```
class Folder {
    friend class Message;
public:
    ~Folder(); // remove self from Messages in msgs
    Folder(const Folder&); // add new folder to each Message in msgs
    Folder& operator=(const Folder&); // delete Folder from lhs messages
                                      // add Folder to rhs messages
    Folder() { } // defaults are ok
    void save(Message&);    // add this message to folder
    void remove(Message&); // remove this message from this folder

    // don't reveal implementation because it's likely to change
    // also better to copy than ref to the internal data structure so
    // have a copy to operate on not the actual contents is safer
    std::vector<Message*> messages(); // return the messages in this folder
    void debug_print(); // print contents and its list of Folders,
private:
    typedef std::set<Message*>::const_iterator Msg_iter;
    std::set<Message*> msgs;  // messages in this folder
```

```
    void copy_msgs(const std::set<Message*>&);// add this Folder to each Message
    void empty_msgs();             // remove this Folder from each Message
    // used by Message class to add self to this Folder's set of Messages
    void addMsg(Message *m) { msgs.insert(m); }
    void remMsg(Message *m) { msgs.erase(m); }
};
```

Exercise 13.19: *Add a* `save` *and* `remove` *operation to the* `Message` *and* `Folder` *classes. These operations should take a* `Folder` *and add (or remove) that* `Folder` *to (from) the* `set` *of* `Folders` *that point to this* `Message`*. The operation must also update the* `Folder` *to know that it points to this* `Message`*, which can be done by calling* `addMsg` *or* `remMsg`*.*

Answer 13.19:

```
void Message::save(Folder &f)
{
    // add f to Folders and this Message to f's list of Messages
    folders.insert(&f);
    f.addMsg(this);
}
void Message::remove(Folder &f)
{
    // remove f from Folders and this Message from f's list of Messages
    folders.erase(&f);
    f.remMsg(this);
}

void Folder::save(Message &m)
{
    // add m and add this folder to m's set of Folders
    msgs.insert(&m);
    m.addFldr(this);
}
void Folder::remove(Message &m)
{
    // erase m from msgs and remove this folder from m
    msgs.erase(&m);
    m.remFldr(this);
}
```

Exercises Section 13.5

Exercise 13.20: *Given the original version of the* `HasPtr` *class that relies on the default definitions for copy-control, describe what happens in the following code:*

```
int i = 42;
HasPtr p1(&i, 42);
HasPtr p2 = p1;
cout << p2.get_ptr_val() << endl;
p1.set_ptr_val(0);
cout << p2.get_ptr_val() << endl;
```

Answer 13.20: After the initialization of `p2` as a copy of `p1`, both objects point to the same object, `i`, and both hold an `int` whose value is 42. The output expression, which calls `p2.get_ptr_val` prints 42.

After the call to `p1.set_ptr_val(0);` the value of the object to which `p1` points is changed to 0. Because both `p1` and `p2` point to the same object; the value of the object to which `p2` points is also changed. Thus, the call to `p2.get_ptr_val` now prints 0.

Exercises Section 13.5.2

Exercise 13.27: *The valuelike* `HasPtr` *class defines each of the copy-control members. Describe what would happen if the class defined*

(a) The copy constructor and destructor but no assignment operator.
(b) The copy constructor and assignment operator but no destructor.
(c) The destructor but neither the copy constructor nor assignment operator.

Answer 13.27:

(a) If the class had a copy constructor and destructor but no assignment operator, then the synthesized assignment operator would copy the pointer in the right-hand operand's `ptr` member. The result would be that two objects would hold the same pointer. When one of those objects is destroyed, the memory to which the pointer points will be freed. Any use of the other object will be undefined—its `ptr` member will point at memory that has been freed.
(b) If the class failed to define a destructor then there would be a memory leak. Each object will have allocated memory in its constructor but that memory will never be freed. Although the assignment operator frees the memory in its left-hand operand, it also dynamically allocates new memory to hold a copy of the data in its right-hand operand. This memory will not be freed if there is no destructor.
(c) If the class uses the synthesized copy constructor and assignment operator, then any time we copy or assign objects of this class we will have objects that point to the same underlying memory. If the class has a destructor that `deletes` the `ptr` member, then destroying an object that shared memory with another object will leave that other object in an undefined state. The remaining object(s) will point to memory that has been freed.

Exercise 13.28: *Given the following classes, implement a default constructor and the necessary copy-control members.*

```
(a) class TreeNode {
    public:
        // ...
    private:
        std::string value;
        int         count;
        TreeNode    *left;
        TreeNode    *right;
    };
```

```
(b) class BinStrTree {
        public:
          // ...
        private:
             TreeNode *root;
        };
```

Answer 13.28:

```
#include <string>
class TreeNode {
    //  private utility functions used by copy control functions
    TreeNode *copy_tree(TreeNode*) const; //  copy the subtree pointed to by this TreeNode
    void free_tree(TreeNode*);            //  walk the subtree freeing each TreeNode
public:
    //  default constructor: initialize pointers to 0
    TreeNode(): count(0), left(0), right(0) { }
    //  copy constructor: copy the subtrees
    TreeNode(const TreeNode &t):
        value(t.value), count(t.count),
        left(copy_tree(t.left)), right(copy_tree(t.right)) { }
    TreeNode& operator=(const TreeNode&);
    ~TreeNode() { free_tree(this); }
    //  ...
```

```
private:
    std::string value;
    int         count;
    TreeNode    *left;
    TreeNode    *right;
};
void TreeNode::free_tree(TreeNode *node)
{
    // once node is 0 we're at the end of the subtree
    if (node == 0) return;

    // walk the subtree to clean up the remaining nodes
    free_tree(node->left); free_tree(node->right);
    // delete the memory for this node
    delete left; delete right;
}

TreeNode *TreeNode::copy_tree(TreeNode *node) const
{
    // stopping condition -- no more copying once we're at the end of the tree
    if (node == 0) return node;
    // there's another node in the tree copy it and its children
    TreeNode *top = new TreeNode(*node);
    top->left = copy_tree(left);
    top->right = copy_tree(right);
    return top;
}
TreeNode &TreeNode::operator=(const TreeNode &rhs)
{
    // check for self-assignment
    if (this == &rhs) return *this;
    // return memory used by the left-hand tree
    free_tree(this);
    // copy the values from the right-hand operand
    value = rhs.value;
    count = rhs.count;
    // copy the subtrees pointed to by the right-hand operand
    left = copy_tree(rhs.left);
    right = copy_tree (rhs.right);
    return *this;
}
/*
 * BinStrTree delegates the work of managing memory to the
 * underlying TreeNode class.  For example, to copy a BinStrTree
 * we copy the TreeNode to which root points, which invokes
 * the TreeNode copy constructor to copy the entire tree. To destroy
 * a BinStrTree, we delete root, which in turn invokes
 * the TreeNode destructor. Etc.
  */
class BinStrTree {
public:
        BinStrTree(): root(0) { } //  initially no nodes in the tree
    // copy a BinStrTree by copying all the nodes in the original
    // the TreeNode copy constructor walks the tree copying the nodes
    BinStrTree(const BinStrTree &t): root(new TreeNode(*t.root)) { }
    // assignment delegates the work to the TreeNode assignment operator
```

```
    // if the root nodes are different in this BinStrTree and rhs,
    // then the assignment of *root from *rhs.root uses the TreeNode assignment operator
    // to delete the existing nodes in this object and copy the nodes from rhs
    BinStrTree &operator=(const BinStrTree &rhs)
                  { *root = *rhs.root; return *this; }
    // uses the TreeNode destructor to walk the tree deleting each node
    ~BinStrTree() { delete root; }
private:
    TreeNode *root;
};
```

Chapter 14

Exercises Section 14.1

Exercise 14.2: *Write declarations for the overloaded input, output, addition and compound-assignment operators for* `Sales_item`*.*

Answer 14.2:

```
class Sales_item {
    friend std::istream& operator>>(std::istream&, Sales_item&);
    friend std::ostream& operator<<(std::ostream&, const Sales_item&);
    Sales_item& operator+=(const Sales_item&);
    // other members as before
};
// nonmember binary operator: must declare a parameter for each operand
Sales_item operator+(const Sales_item&, const Sales_item&);
```

Exercise 14.4: *Both the* `string` *and* `vector` *types define an overloaded* `==` *that can be used to compare objects of those types. Identify which version of* `==` *is applied in each of the following expressions:*

```
string s; vector<string> svec1, svec2;
"cobble" == "stone"
svec1[0] == svec2[0];
svec1 == svec2
```

Answer 14.4:

`"cobble" == "stone"` Both operands are character string literals. The operands types therefore are `const char*`. The built-in `==` operator is used and compares the address values of these literals. This behavior is almost certainly not what was intended.

`svec1[0] == svec2[0]` The operands are the values returned by the `vector` subscript operator. That return is a reference to the element type of the `vector`, which in this case is `string`. Hence, the `string` `==` operator is used to compare the operands.

`svec1 == svec2` Here the operands are both `vector`s. The `vector` `==` is used.

Exercises Section 14.1.1

Exercise 14.6: *Explain why and whether each of the following operators should be class members:*
(a) `+` *(b)* `+=` *(c)* `++` *(d)* `->` *(e)* `<<` *(f)* `&&` *(g)* `==` *(h)* `()`

Answer 14.6:

(a) + ordinarily should be a nonmember operator function. Making it a nonmember allows conversions to be applied to both operands. If it is a member, then conversions would be allowed on the right-hand operand but not the left.
(b) += changes the state of the left-hand operand and so ordinarily += is defined as a member of the class. ++ also changes the state of the left-hand operand. It usually is defined as a member function.
(c) -> must be a member of its class.
(d)
(e) << is generally used as the output operator. When used to perform output its left-hand operand is an `ostream&`. We cannot add our own operators to this class and so, when used to do output, << is a nonmember function.

Alternatively, if << is defined by a class for some other purpose, then whether it should be a member or nonmember depends on what the operator does. If its behavior mimics the built-in << operator, then the overloaded << would change the value of its left-hand operand. If the operator changes the object it should be a member. Alternatively, if << is used as a simple binary operator that does not affect its operands, then it probably should be defined as a nonmember, allowing either operand to be converted.
(f) && usually should not be overloaded. The short-circuit operand evaluation properties of && are not preserved. Mistakes by users assuming that short-circuit evaluation is performed are likely when a class overloads &&. However, if the operator is overloaded, whether it is a member or not is a design choice. As with other binary operators, if the overloaded definition changes the left-hand operand then it should be a member of that class. Otherwise, it should be a nonmember.
(g) == like + is usually defined as a nonmember, allowing conversions on both operands. A class that defines == should use that operator only for an operation that determines whether two objects are equal. The library containers and algorithms uses the == operator assuming that it implements an equality relationship.
(h) () must be a member of its class.

Exercises Section 14.2.1

Exercise 14.7: *Define an output operator for the following* `CheckoutRecord` *class:*

```
class CheckoutRecord {
public:
    // ...
private:
    double book_id;
    string title;
    Date date_borrowed;
    Date date_due;
    pair<string,string> borrower;
    vector< pair<string,string>* > wait_list;
};
```

Answer 14.7:

```
#include <iostream>
#include <string>
#include <vector>
#include <utility>
#include <cstddef>
// simple Date structure to allow us to print the contents
struct Date {
    std::size_t year;
    std::size_t month;
    std::size_t day;
};
class CheckoutRecord {
```

```
    // add friend declaration to allow access to member data
    friend std::ostream& operator<<(std::ostream&, const CheckoutRecord&);
public:
    // interface functions here
private:
    double book_id;
    std::string title;
    Date date_borrowed;
    Date date_due;
    std::pair<std::string,std::string> borrower;
    // note the wait_list holds pointers to dynamically allocated pairs
    std::vector< std::pair<std::string,std::string>* > wait_list;
};
using std::endl; using std::ostream;
using std::string; using std::pair; using std::vector;
ostream& operator<<(ostream &os, const Date &date)
{
    os << date.month << "/" << date.day << "/" << date.year;

    return os;
}
ostream& operator<<(ostream &os, const CheckoutRecord &record)
{
    // print data members without any formatting
    os << record.book_id << " " << record.title
       << " " << record.date_borrowed << " " << record.date_due
       << " " << record.borrower.first
       << " " << record.borrower.second << endl;
    // print the wait list on a new line, indented
    vector< pair<string,string>* >::const_iterator iter
                                               = record.wait_list.begin();
    while (iter != record.wait_list.end()) {
        // print a tab at the start of the wait_list
        if (iter == record.wait_list.begin())
            os << "\t";
        // iter refers to a pointer to a pair<string, string>
        // to make the syntax easier, p is the underlying object
        pair<string, string> p = *(*iter);
        os << p.first << " " << p.second;

        // print a space after all but the last element
        if (++iter != record.wait_list.end())
            os << " ";
    }
    // return the stream we just wrote to
    return os;
}
```

Exercises Section 14.2.2

Exercise 14.9: *Describe the behavior of the* `Sales_item` *input operator if given the following input:*

(a) `0-201-99999-9 10 24.95`
(b) `10 24.95 0-210-99999-9`

Answer 14.9:

(a) The input operator should successfully read this transaction, assigning `0-201-99999-9` to the `isbn` member, 10 to the units_sold and 249.50 to the revenue member.

(b) The Sales_item input operator will also successfully read this transaction, but the Sales_item object that results will be nonsensical, as is this input. The operator executes by first reading 10 into the isbn field. Next it reads an int into the units_sold member. The int it reads will be 24—the read into units_sold reads up to the first nonnumeric character. Next it reads a double so it will read .95 into price. It will multiply .95 * 24 to generate the revenue member. The resulting Sales_item thus has an isbn of 10, a units_sold of 24 and revenue of 22.80 (e.g., 24 * .95).

It is worth noting that the file is left positioned so that the next read will see 0-210-99999-9.

Exercise 14.11: *Define an input operator for the* CheckoutRecord *class defined in the exercises for Section 14.2.1 (p. 515). Be sure the operator handles input errors.*

Answer 14.11:

```
class CheckoutRecord {
    friend std::istream& operator>>(std::istream&, CheckoutRecord&);
    // remainder of the class is unchanged
public:
    // typedefs to make dealing with the wait_list easier
    typedef std::vector< std::pair<std::string,std::string>* > names;
    typedef names::size_type size_type;
private:
    // utility function to free the pairs on the wait_list
    void free_wait_list() {
        for (size_type i = 0; i != wait_list.size(); ++i)
                delete wait_list[i];  // free each pair
        wait_list.clear();  // now reset the vector to empty
    }
};
#include <stdexcept>
#include <sstream>
using std::runtime_error; using std::istream; using std::istringstream;
istream& operator>>(istream &is, Date &date)
{
    is >> date.month >> date.day >> date.year;

    return is;
}
// Input operator will generate an empty CheckoutRecord if there are problems.
// We assume the input data is in the same format as the record itself
// and that the entire record, including the wait list is on a single line.
istream& operator>>(istream &is, CheckoutRecord &record)
{
    // read the entire line and then use an istringstream
    // to break it up into separate data elements
    // must read the line because the wait_list could be empty
    string line;
    getline(is, line);
    istringstream input_line(line);
    // read simple data members first
    input_line >> record.book_id >> record.title
               >> record.date_borrowed >> record.date_due
               >> record.borrower.first >> record.borrower.second;
    // remember to free the existing wait_list
    // no need to check for input failures yet; the old wait_list
    // needs to be freed regardless of whether input fails
```

```
    record.free_wait_list();
    // now loop through the data to create new wait_list if any
    string first, last;
    while (input_line >> first >> last) {
        // allocate a new pair for the wait_list
        pair<string,string> *pair_ptr =
            new pair<string,string>(first,last);
        record.wait_list.push_back(pair_ptr);
    }
    // Before returning, check whether an input operation failed.
    // If so, overwrite record so it is in a consistent state
    if (!is || !input_line) {
        // but leave the input stream in whatever error condition
        // caused the problem. Leaving the stream state unchanged
        // will let our users figure out what to do next
        record = CheckoutRecord();
    }
    return is;
}
```

Exercises Section 14.3

Exercise 14.12: *Write the* `Sales_item` *operators so that* + *does the actual addition and* += *calls* +*. Discuss the disadvantages of this approach compared to the way the operators were implemented in this section.*

Answer 14.12:

```
// assumes that both objects refer to the same isbn
Sales_item& Sales_item::operator+=(const Sales_item& rhs)
{
    *this = *this + rhs;    // delegate real work to the + operator
    return *this;
}
// assumes that both objects refer to the same isbn
// Note: This version of assignment must be a friend of Sales_item
Sales_item
operator+(const Sales_item& lhs, const Sales_item& rhs)
{
    Sales_item ret;  // define a local object that we'll return
    // add components from lhs and rhs and store in ret
    ret.units_sold = lhs.units_sold + rhs.units_sold;
    ret.revenue = lhs.revenue + rhs.revenue;
    // set the isbn
    ret.isbn = lhs.isbn;
    return ret;             // return a copy of ret
}
```

This version is less efficient than the version presented in the text. The problem is that the + operator creates a local object of type `Sales_item` to hold the sum of its two operands. That object is then copied as the return value from the function. When the `+=` operation runs, it calls `+`, which generates this unnamed temporary that is then used to overwrite the existing `Sales_item` to which `this` points. Doing the work in this order is wasteful, as it would be safe and easier to just add the `rhs` value directly into the `*this` object.

Exercises Section 14.4

Exercise 14.14: *Define a version of the assignment operator that can assign an* `isbn` *to a* `Sales_item`*.*

Answer 14.14:

```
class Sales_item {
public:
    Sales_item &operator=(const std::string&);
    // other members and friends as before
};
// reset the object to the default state and set isbn to the right-hand operand
Sales_item &Sales_item::operator=(const string &rhs)
{
    isbn = rhs;
    units_sold = 0;
    revenue = 0;
    return *this;
}
```

Exercise 14.15: *Define the class assignment operator for the* `CheckoutRecord` *introduced in the exercises to Section 14.2.1 (p. 515).*

Answer 14.15:

```
class CheckoutRecord {
public:
    CheckoutRecord &operator=(const CheckoutRecord&);
    // other members and friends as before
public:
    // typedefs to make dealing with the wait_list easier
    typedef std::vector< std::pair<std::string,std::string>* > names;
    typedef names::size_type size_type;
    typedef std::pair<std::string,std::string> name_pair;
private:
    // utility function to copy pairs from another wait_list
    void copy_wait_list(const CheckoutRecord &record) {
        for (size_type i = 0; i != record.wait_list.size(); ++i) {
            // allocate the new pair and put it on the wait_list
            wait_list.push_back(new name_pair(*record.wait_list[i]));
        }
    }
};
CheckoutRecord &CheckoutRecord::operator=(const CheckoutRecord &rhs)
{
    // as usual check for self-assignment first
    if (this == &rhs) return *this;
    // now assign components from rhs to this object
    book_id = rhs.book_id;
    title = rhs.title;
    date_borrowed = rhs.date_borrowed;
    date_due = rhs.date_due;
    borrower = rhs.borrower;
    // next free the existing wait_list
    free_wait_list();
    // and copy the wait list from the rhs
    copy_wait_list(rhs);
    return *this;
}
```

Exercises Section 14.5

Exercise 14.17: *Define a subscript operator that returns a name from the waiting list for the* `CheckoutRecord` *class from the exercises to Section 14.2.1 (p. 515).*

Answer 14.17:

```
class CheckoutRecord {
public:
    name_pair &operator[](size_type);
    // other members and friends as before
};
#include <stdexcept>
using std::out_of_range;
CheckoutRecord::name_pair &
CheckoutRecord::operator[](size_type index)
{
    if (index > wait_list.size())
        throw out_of_range("CheckoutRecord index out of range");
    return *wait_list[index];
}
```

Exercises Section 14.6

Exercise 14.20: *In our sketch for the* `ScreenPtr` *class, we declared but did not define the assignment operator. Implement the* `ScreenPtr` *assignment operator.*

Answer 14.20:

```
ScreenPtr& ScreenPtr::operator=(const ScreenPtr &rhs)
{
    ++rhs.ptr->use;       // increment use count on rhs first
    if (--ptr->use == 0)
         delete ptr;      // if use count goes to 0 on this object, delete it
    ptr = rhs.ptr;        // copy the ScrPtr object
    return *this;
}
```

Exercise 14.21: *Define a class that holds a pointer to a* `ScreenPtr`*. Define the overloaded arrow operator for that class.*

Answer 14.21: As with the other overloaded arrow operators we defined, the overloaded arrow for this class should return the pointer it holds. This operator returns a `ScreenPtr*`:

```
class ScreenPtrPtr {
    ScreenPtr *p;  // points to a ScreenPtr
public:
    // other operations
    ScreenPtr *operator->() { return p; }
    const ScreenPtr *operator->() const { return p; }
};
```

When a user executes this operator, the result will be a class that defines an overloaded arrow. That operator in turn will obtain the underlying pointer to a `Screen`.

Exercises Section 14.7

Exercise 14.23: *The class* `CheckedPtr` *represents a pointer that points to an array of* `int`*s. Define an overloaded subscript and dereference for this class. Have the operator ensure that the* `CheckedPtr` *is valid: It should not be possible to dereference or index one past the end of the array.*

Answer 14.23:

```
class CheckedPtr {
public:
    // subscript operator
    int &operator[](const std::size_t);
    const int &operator[](const std::size_t) const;
    // dereference operator
    int& operator*();
    const int& operator*() const;

    // other members as before
};

int &CheckedPtr::operator[](const size_t index)
{
    // check whether index greater than or equal to the number of elements
    if (index >= end - beg)
        throw out_of_range("dereference past the end");
    return beg[index];
}
const int &CheckedPtr::operator[](const size_t index) const
{
    // check whether index greater than or equal to the number of elements
    if (index >= end - beg)
        throw out_of_range("dereference past the end");
    return beg[index];
}
int& CheckedPtr::operator*()
{
    if (curr == end)
        throw out_of_range("dereference past the end");
    return *curr;
}
const int& CheckedPtr::operator*() const
{
    if (curr == end)
        throw out_of_range("dereference past the end");
     return *curr;
}
```

Exercise 14.24: *Should the dereference or subscript operators defined in the previous exercise also check whether an attempt is being made to dereference or index one before the beginning of the array? If not, why not? If so, why?*

Answer 14.24: The parameter to the subscript operator is a `size_t`, which is an `unsigned` value. Because the argument is `unsigned` we know that it cannot be less than zero. Because the lowest valid index in the array is also zero, there is no need to check whether the index value it too small.

Exercise 14.25: *To behave like a pointer to an array, our* `CheckedPtr` *class should implement the equality and relational operators to determine whether two* `CheckedPtr`*s are equal, or whether one is less-than another, and so on. Add these operations to the* `CheckedPtr` *class.*

Answer 14.25:

```
class CheckedPtr {
    friend bool operator==(const CheckedPtr&, const CheckedPtr&);
    friend bool operator!=(const CheckedPtr&, const CheckedPtr&);
    friend bool operator<(const CheckedPtr&, const CheckedPtr&);
    friend bool operator<=(const CheckedPtr&, const CheckedPtr&);
    friend bool operator>(const CheckedPtr&, const CheckedPtr&);
    friend bool operator>=(const CheckedPtr&, const CheckedPtr&);
};
```

```
// two CheckedPtrs are equal if they point to the same place in the same array
bool operator==(const CheckedPtr &lhs, const CheckedPtr &rhs)
{
    return lhs.beg == rhs.beg
            && lhs.end == rhs.end
            && lhs.curr == rhs.curr;
}
bool operator!=(const CheckedPtr &lhs, const CheckedPtr &rhs)
{
    return !(lhs == rhs);
}
// relational operators only apply if the CheckedPtrs refer to the same array
// If not, throw an exception
bool operator<(const CheckedPtr &lhs, const CheckedPtr &rhs)
{
    if (lhs.beg != rhs.beg || lhs.end != rhs.end)
        throw out_of_range("attempt to compare incomensurate CheckedPtrs");
    return lhs.curr < rhs.curr;
}
/*
 * each of the other relationals looks the same:
 * First check that the arrays are the same, throwing exception if not
 * Next apply the appropriate operation.
 * <= shown here, others left for the reader
 */
bool operator<=(const CheckedPtr &lhs, const CheckedPtr &rhs)
{
    if (lhs.beg != rhs.beg || lhs.end != rhs.end)
        throw out_of_range("attempt to compare incomensurate CheckedPtrs");
    return lhs.curr <= rhs.curr;
}
```

Exercise 14.28: *We did not define a `const` version of the increment and decrement operators. Why?*

Answer 14.28: There is no need to define a `const` version of these operators because increment and decrement change the underlying object to which they are applied. However, a `const` object may not be changed and so there is no sensible definition for a `const` version of the increment or decrement operators.

Exercise 14.29: *We also didn't implement arrow. Why?*

Answer 14.29: We didn't implement the arrow operator because the underlying array holds `int`s. There is no arrow operator for `int`.

Exercises Section 14.8

Exercise 14.31: *Define a function object to perform an if-then-else operation: The function object should take three parameters. It should test its first parameter and if that test succeeds, it should return its second parameter, otherwise, it should return its third parameter.*

Answer 14.31:

```
class condObj {
public:
    condObj(bool i, int j, int k): op1(i), op2(k), op3(j) { }
    int operator()() { return op1 ? op2 : op3; }
private:
    bool op1;
    int op2, op3;
};
```

Exercises Section 14.8.1

Exercise 14.35: *Write a class similar to* `GT_cls`*, but that tests whether the length of a given* `string` *matches its bound. Use that object to rewrite the program in Section 11.2.3 (p. 400) to report how many words in the input are of sizes 1 through 10 inclusive.*

Answer 14.35:

```
// determine whether a length of a given word is equal to a stored bound
class EQ {
public:
    EQ(size_t val = 0): bound(val) { }
    bool operator()(const string &s) { return s.size() == bound; }
private:
    size_t bound;
};
using std::greater; using std::bind2nd;
int main()
{
    vector<string> words;
    // copy contents of each book into a single vector
    string next_word;
    while (cin >> next_word) {
        // insert next book's contents at end of words
        words.push_back(next_word);
    }
    // sort words alphabetically so we can find the duplicates
    sort(words.begin(), words.end());
    /* eliminate duplicate words:
     * unique reorders words so that each word appears once in the
     *      front portion of words and returns an iterator
     *      one past the unique range;
     * erase uses that iterator as the beginning of the range
     *      to erase, after erase only the unique words remain
     */
    vector<string>::iterator last_word =
                unique(words.begin(), words.end());
    words.erase(last_word, words.end());
    // sort words by size, but maintain alphabetic order for words of the same size
    stable_sort(words.begin(), words.end(), isShorter);
    // loop through all sizes up to the size of the longest string in words
```

```
    // use the reverse iterator rbegin to get the size of the last string
    string::size_type maxlen = words.rbegin()->size();
    for (size_t i = 1; i <= maxlen; ++i)
        cout << count_if(words.begin(), words.end(), EQ(i))
             << " words " << i
             << " characters long" << endl;
    return 0;
}
```

Exercise 14.36: *Revise the previous program to report the count of words that are sizes 1 through 9 and 10 or more.*

Answer 14.36:

```
// first count how many are size 1 through 9
size_t cnt = 0;
for (size_t i = 0; i != 10; ++i)
    cnt += count_if(words.begin(), words.end(), EQ(i));
cout << cnt << " words from 1 through 9 characters" << endl;
// next print how many are size 10 or more
cout << count_if(words.begin(), words.end(), GT_cls(10))
     << " words 10 characters or longer" << endl;
```

Exercises Section 14.8.3

Exercise 14.37: *Using the library function objects and adaptors, define an object to:*

(a) Find all values that are greater than 1024.
(b) Find all strings that are not equal to `pooh`*.*
(c) Multiply all values by 2.

Answer 14.37:

```
// function object to test whether a value is greater than 1024
bind2nd(greater<int>(), 1024)
// function object to test whether a string is not equal to pooh
bind1st(not_equal_to<string>(), string("pooh"))
// function object to multiply a value by 2
bind1st(multiplies<int>(), 2)
```

Exercises Section 14.9.2

Exercise 14.41: *Explain the difference between these two conversion operators:*

```
class Integral {
public:
    const int();
    int() const;
};
```

Are either of these conversions too restricted? If so, how might you make the conversion more general?

Answer 14.41: The first operator converts an `Integral` object to a `const int`. It may only be used on nonconst `Integral` objects. The second converts any kind of `Integral` (`const` or nonconst) to a plain `int`. The second conversion operator is more general, because it can be applied to `const` or nonconst objects. Moreover, because both operators return by copying the return value there is little (or no) benefit in defining the conversion operator to return a `const int` instead of a plain `int`. In all ways, the second operator seems preferable.

Exercises Section 14.9.4

Exercise 14.44: *Show the possible class-type conversion sequences for each of the following initializations. What is the outcome of each initialization?*

```
class LongDouble {
    operator double();
    operator float();
};
LongDouble ldObj;
(a) int ex1 = ldObj;        (b) float ex2 = ldObj;
```

Answer 14.44: Note that these conversion operators must be made `public` in order for object initializations to rely on them. Given that the conversions are accessible, then:

(a) The first initialization is ambiguous. Either conversion could be used: `ldObj` could be converted to `double` and then a standard conversion applied to convert that `double` to `int`. Alternatively, `ldObj` could be converted to `float` and then a standard conversion applied to convert that `float` to `int`.
(b) In the second expression there is a direct conversion sequence: `ldObj` can be converted directly to the type of `ex2` by using the `operator float` conversion operator.

Exercises Section 14.9.5

Exercise 14.46: *Which* `operator+`*, if any, is selected as the best viable function for the addition operation in* `main`*? List the candidate functions, the viable functions, and the type conversions on the arguments for each viable function.*

```
class Complex {
    Complex(double);
    // ...
};
class LongDouble {
    friend LongDouble operator+(LongDouble&, int);
public:
    LongDouble(int);
    operator double();
    LongDouble operator+(const complex &);
    // ...
};
LongDouble operator+(const LongDouble &, double);

LongDouble ld(16.08);
double res = ld + 15.05; // which operator+ ?
```

Answer 14.46: The addition is ambiguous. The viable functions are:

- The built-in addition operator taking two `double` arguments. This operator is viable because the `operator double` in class `LongDouble` could be used to convert `ld` to `double`.

- The addition operator that is a member of `LongDouble` is viable. That operator requires a left-hand operand of type `LongDouble` and a right-hand operand of type `Complex`. In this expression the type of the left-hand operand is `LongDouble` and the type of the right-hand operand is `double`. Because the `Complex` class defines a nonexplicit constructor that takes a `double` we can convert the right-hand operand to the necessary type.

- The nonmember function `operator+(LongDouble&, int)` is viable. Its first parameter is an exact match for the left-hand operand and there is a conversion from `double` to `int` for the second operant.

- The nonmember function `operator+(const LongDouble&, double)` is viable. Its first parameter is a reference to a `const LongDouble` and the second parameter is a `double`. The second parameter is an exact match for the right-hand operand, but the left-hand operand requires a conversion. The left-hand operand is a plain (non`const`) `LongDouble` and the parameter type is a reference to a `const LongDouble`.

This call is ambiguous because any of these functions are equally good matches:

- The nonmember function `operator+(const LongDouble&, double)` is an exact match on the second operand and requires a built-in conversion for the first.
- The other nonmember function, `operator+(LongDouble&, int)`, is an exact match for the first operand but requires a built-in conversion for the second operand.
- The member function `LongDouble operator+(const complex&)` is an exact match on the left-hand operand but requires a class-defined conversion on the right-hand operand.
- The built-in `double` addition operator is an exact match for the right-hand operand but requires a class conversion for the left-hand operand.

Chapter 15

Exercises Section 15.2.1

Exercise 15.4: *A library has different kinds of materials that it lends out—books, CDs, DVDs, and so forth. Each of the different kinds of lending material has different check-in, check-out, and overdue rules. The following class defines a base class that we might use for this application. Identify which functions are likely to be defined as virtual and which, if any, are likely to be common among all lending materials. (Note: we assume that* `LibMember` *is a class representing a customer of the library, and* `Date` *is a class representing a calendar day of a particular year.)*

```
class Library {
public:
    bool check_out(const LibMember&);
    bool check_in (const LibMember&);
    bool is_late(const Date& today);
    double apply_fine();
    ostream& print(ostream& = cout);
    Date due_date() const;
    Date date_borrowed() const;
    string title() const;
    const LibMember& member() const;
};
```

Answer 15.4: The functions that handle `Date`s and `LibMember`s—the members `is_late`, `due_date`, `date_borrowed`, and `member`—are likely not to vary by type of lending material. There is likely no need to define these as virtual functions.

Those related to the material, such as `check_out` and `check_in`, will almost surely vary by type of lending material. Similarly, the `print` function is likely to need to print information that is specific to each kind of lending material. These three functions probably should be virtual.

Whether the remaining two functions, `title` and `is_late`, should be virtual depends on the nature of the lending material and the policies of the library. We can guess that `title` need not be virtual—guessing this way implies that all kinds of materials have titles and that the use of the `title` doesn't vary by type of lending material. We can also guess that `is_late` would not change—it seems likely that `is_late` is a function of the current date and the `due_date`. However, it is possible that certain kinds of materials might have a grace period that others do not share. Until we know more about the library policies it would be hard to determine whether this function should be virtual.

Exercises Section 15.2.3

Exercise 15.5: *Which of the following declarations, if any, are incorrect?*

```
class Base { ... };

(a) class Derived : public Derived { ... };
(b) class Derived : Base { ... };
(c) class Derived : private Base { ... };
(d) class Derived : public Base;
(e) class Derived inherits Base { ... };
```

Answer 15.5:

(a) Incorrect: a class may not be derived from itself.
(b) OK: the access label is optional. Section 15.2.5 (p. 570) will describe what happens when the access label is omitted.
(c) OK: `Derived` inherits from `Base` but that fact is not part of the interface to `Derived`.
(d) Incorrect: This is a declaration not a definition of class `Derived`. The derivation list is included only in the definition of a class, not its declaration.
(e) Incorrect: The word "inherits" has no meaning in C++.

Exercise 15.7: *We might define a type to implement a limited discount strategy. This class would give a discount for books purchased up to a limit. If the number of copies purchased exceeds that limit, then the normal price should be applied to any books purchased beyond the limit. Define a class that implements this strategy.*

Answer 15.7:

```
// discount (a fraction off list) for only a specified number of copies,
// additional copies sold at standard price
class Lim_item : public Item_base {
public:
    // redefines base version so as to implement limited discount policy
    double net_price(std::size_t) const;
private:
    std::size_t max_qty;    // maximum number sold at discount
    double discount;        // fractional discount to apply
};
```

```
// use discounted price for up to a specified number of items
// additional items priced at normal, undiscounted price
double Lim_item::net_price(size_t cnt) const
{
    size_t discounted = min(cnt, max_qty);
    size_t undiscounted = cnt - discounted;
    return discounted * (1 - discount) * price
           + undiscounted * price;
}
```

Exercises Section 15.2.4

Exercise 15.8: *Given the following classes, explain each* `print` *function:*

```
struct base {
   string name() { return basename; }
   virtual void print(ostream &os) { os << basename; }
private:
   string basename;
};

struct derived {
   void print() { print(ostream &os); os << " " << mem; }
private:
   int mem;
};
```

If there is a problem in this code, how would you fix it?

Answer 15.8: There are two bugs in this exercise as presented in the text that will be printed in the fourth and subsequent printings. The definition of `derived::print` that was presented as

```
   void print() { print(ostream &os); os << " " << mem; }
```

should have been written as:

```
   void print(ostream &os) { print(os); os << " " << mem; }
```

The other problem is that, although the exercise assumes that `derived` inherits from `base`, the classes as presented are actually unrelated. The definition of `derived` should be changed to indicate that it inherits from `base`:

```
struct derived: base {
```

With these fixes, the code can be explained as follows: The `print` function in class `base` prints its `string` member. The `print` function in `derived` intends to call its base-class `print` member to print the `base::basemem` of the `derived` object. However, the call as written is a `virtual` call that will (repeatedly) call the `print` member in the derived.

The `print` function in `derived` should have been written as:

```
   void print(ostream &os) { base::print(os); os << " " << mem; }
```

This version correctly calls the version of `print` from `base`, which prints `basemem`. Having printed the `base` part, the `derived` `print` function next prints the `mem` member of the `derived` object.

Exercise 15.9: *Given the classes in the previous exercise and the following objects, determine which function is called at run time:*

```
base bobj;     base *bp1 = &base;   base &br1 = bobj;
derived dobj;  base *bp2 = &doboj;  base &br2 = dobj;

(a) bobj.print();  (b) dobj.print();  (c) bp1->name();
(d) bp2->name();   (e) br1.print();   (f) br2.print();
```

Answer 15.9: Assuming the class definitions are fixed as described in the previous exercise, then:

(a) `bobj.print()` calls `base::print`—`bobj` is an object. When we call a virtual function through an object, the call is resolved at compile time and is the version defined by the type of the object.
(b) `dobj.print()` calls `derived::print`—`dobj` is an object, so the call is resolved at compile time to call the version defined by the type of `dobj`.

(c) `bp1->name()` calls `base::name`. The `name` function is nonvirtual so this call is resolved at compile time. Which `name` function is called is determined based on the type of the object, reference or pointer through which the call is made. In this case, `bp1` is a pointer to `base`, which means that the `name` function defined in class `base` is called.

(d) `bp2->name()` calls `base::name`. The `name` function is nonvirtual so this call is resolved at compile time and is based on the type of the object, reference or pointer through which the function is called. The function is called through `bp2`, which is a pointer to `base`. The fact that the pointer points to a `derived` object is irrelevant.

(e) `br1.print()` calls `base::print`. Because `print` is virtual and this call is made through a reference, the decision as to which version of `print` to call is made at runtime and is based on the type of the object to which the reference refers. In this case, we know that `br1` refers to a `base` object and so the call is resoved to `base::print`.

(f) `br2.print()` calls `derived::print`. Again, `print` is virtual and the call is made through a reference and so the call is resolved at runtime. In this case, we know that `br2` refers to a `derived` object and so the call is resoved to `derived::print`.

Exercises Section 15.2.5

Exercise 15.10: *In the exercises to Section 15.2.1 (p. 562) you wrote a base class to represent the lending policies of a library. Assume the library offers the following kinds of lending materials, each with its own check-out and check-in policy. Organize these items into an inheritance hierarchy:*

```
book                  audio book           record
children's puppet     sega video game      video
cdrom book            nintendo video game  rental book
sony playstation video game
```

Answer 15.10:

```
class Library {
public:
    virtual bool check_out(const LibMember&);
    virtual bool check_in (const LibMember&);
    virtual std::ostream& print(std::ostream& = std::cout);
    Date due_date() const;
    Date date_borrowed() const;
    bool is_late(const Date& today);
    double apply_fine();
    std::string title() const;
    const LibMember& member() const;
private:
    std::string item_name;
    LibMember borrower;
    Date due;
    Date when_borrowed;
};
class Book: public Library {
public:
    bool check_out(const LibMember&);
    bool check_in (const LibMember&);
    std::ostream& print(std::ostream& = std::cout);
};
class Cdrom_book: public Book {
public:
    bool check_out(const LibMember&);
    bool check_in (const LibMember&);
```

```
    std::ostream& print(std::ostream& = std::cout);
};
class Audio_book: public Book {
public:
    bool check_out(const LibMember&);
    bool check_in (const LibMember&);
    std::ostream& print(std::ostream& = std::cout);
};
class Rental_book: public Book {
public:
    bool check_out(const LibMember&);
    bool check_in (const LibMember&);
    std::ostream& print(std::ostream& = std::cout);
};
class Game: public Library {
public:
    bool check_out(const LibMember&);
    bool check_in (const LibMember&);
    std::ostream& print(std::ostream& = std::cout);
};
class Sega: public Game {
public:
    bool check_out(const LibMember&);
    bool check_in (const LibMember&);
    std::ostream& print(std::ostream& = std::cout);
};
class Sony: public Game {
public:
    bool check_out(const LibMember&);
    bool check_in (const LibMember&);
    std::ostream& print(std::ostream& = std::cout);
};
class Nintendo: public Game {
public:
    bool check_out(const LibMember&);
    bool check_in (const LibMember&);
    std::ostream& print(std::ostream& = std::cout);
};
class Video: public Library {
public:
    bool check_out(const LibMember&);
    bool check_in (const LibMember&);
    std::ostream& print(std::ostream& = std::cout);
};
class Audio: public Library {
public:
    bool check_out(const LibMember&);
    bool check_in (const LibMember&);
    std::ostream& print(std::ostream& = std::cout);
};
```

Exercises Section 15.2.7

Exercise 15.13: *Given the following classes, list all the ways a member function in* `C1` *might access the* `static` *members of* `ConcreteBase`*. List all the ways an object of type* `C2` *might access those members.*

```
struct ConcreteBase {
    static std::size_t object_count();
protected:
    static std::size_t obj_count;
};
struct C1 : public ConcreteBase { /* . . . */ };
struct C2 : public ConcreteBase { /* . . . */ };
```

Answer 15.13: Because these members are either public or protected, a member function in `C1` can access both static members of `ConcreteBase`. A member function could do so directly from `ConcreteBase` class, such as `ConcreteBase::obj_count` or through the member function's own class, for example by writing `C1::obj_count`. The member function could also access them through a a pointer, object or reference of type `C1` using normal dot or arrow operators.

An object of type `C2` could access only the public static members of the base class. It could do so through its class `C2::object_count()` or through its base class `ConcreteBase::object_count()` or using the dot operator `obj.object_count()`. A pointer to the object could use the arrow operator.

Exercises Section 15.4.2

Exercise 15.15: *Identify the base- and derived-class constructors for the library class hierarchy described in the first exercise on page 575.*

Answer 15.15: Given the description of the interface so far, we'll infer that each `Library` object represents an actual, physical item (a book, a CD, a tape, etc.) that belongs to the library. Further, we'll assume that each item might or might not be lent out at any given time. The constructor will initialize a data field that holds the item's name but any data associated with the `LibMember` or dates lent or due would be set by members that withdraw or return the item. Hence, the base class will have a single constructor that might look something like:

```
class Library {
public:
    // no default constructor: each Library object must
    // have a title; other members set to default values
    // and other member functions will set them when the item
    // is withdrawn or returned
    Library(const std::string &t): item_name(t) { }
    virtual ~Library() { }
};
```

This class will not have a default constructor. We must provide the item's name when creating a `Library` object.

Each derived class must initialize that name and so must also have a constructor that takes a `string`. The constructors might look something like:

```
class Book: public Library {
public:
    Book(const std::string &t): Library(t) { }
};
class Cdrom_book: public Book {
public:
    Cdrom_book(const std::string &t): Book(t) { }
};
class Audio_book: public Book {
```

```
public:
    Audio_book(const std::string &t): Book(t) { }
};
class Rental_book: public Book {
public:
    Rental_book(const std::string &t): Book(t) { }
};
class Game: public Library {
public:
    Game(const std::string &t): Library(t) { }
};
class Sega: public Game {
public:
    Sega(const std::string &t): Game(t) { }
};
class Sony: public Game {
public:
    Sony(const std::string &t): Game(t) { }
};
class Nintendo: public Game {
public:
    Nintendo(const std::string &t): Game(t) { }
};
class Video: public Library {
public:
    Video(const std::string &t): Library(t) { }
};
class Audio: public Library {
public:
    Audio(const std::string &t): Library(t) { }
};
```

Exercises Section 15.4.4

Exercise 15.17: *Describe the conditions under which a class should have a virtual destructor.*

Answer 15.17: A class in an inheritance hierarchy needs a virtual destructor if it is ever possible that a pointer to a derived object might be deleted through a pointer to the base type. In general, this requirement means that a class that serves as a base class should define a virtual destructor. Often such destructors are empty—they do no work.

Unless there is work for the destructor to do, it is sufficient for the root base class to define the destructor. Once the base class defines its destructor as virtual, the fact that the destructor is virtual is inherited by the derived classes. There is no need for the derived classes to define their own virtual destructors.

Exercise 15.18: *What operations must a virtual destructor perform?*

Answer 15.18: The fact that a destructor is virtual has no effect on the work that the destructor performs. The operations a destructor must perform depend entirely on the details of the class design. The destructor cleans up an object, releasing any resources that were allocated during the object's lifetime. Many classes have no work for a destructor to do—there are no resources allocated that have to be cleaned up, nor any other actions that must be taken when an object is destroyed.

A class needs a virtual destructor if it is possible for an object of a derived type to be destroyed through a pointer to a base type. In such cases the destructor is often empty: If there is no work for the destructor to do, the destructor may be defined as virtual and have an empty function body.

Exercises Section 15.5.1

Exercise 15.22: *Redefine* `Bulk_item` *and the class you implemented in the exercises from Section 15.2.3 (p. 567) that represents a limited discount strategy to inherit from* `Disc_item`*.*

Answer 15.22:

```
// class to hold discount rate and quantity
// derived classes will implement pricing strategies using these data
class Disc_item : public Item_base {
public:
    Disc_item(const std::string& book = "",
              double sales_price = 0.0,
              std::size_t qty = 0, double disc_rate = 0.0):
                 Item_base(book, sales_price),
                 quantity(qty), discount(disc_rate) { }
protected:
    std::size_t quantity;  // purchase size for discount to apply
    double discount;       // fractional discount to apply
};

// discount kicks in when a specified number of copies of same book are sold
// the discount is expressed as a fraction to use to reduce the normal price
class Bulk_item : public Disc_item {
public:
    Bulk_item(const std::string& book = "",
              double sales_price = 0.0,
              std::size_t qty = 0, double disc_rate = 0.0):
          Disc_item(book, sales_price, qty, disc_rate) { }
    // redefines base version so as to implement bulk purchase discount policy
    double net_price(std::size_t) const;
};

// if specified number of items are purchased, use discounted price
double Bulk_item::net_price(size_t cnt) const
{
    if (cnt >= min_qty)
        return cnt * (1 - discount) * price;
    else
        return cnt * price;
}
```

Exercises Section 15.5.2

Exercise 15.23: *Given the following base- and derived-class definitions*

```
struct Base {
    foo(int);
protected:
    int bar;
    double foo_bar;
};

struct Derived : public Base {
    foo(string);
    bool bar(Base *pb);
    void foobar();
protected:
    string bar;
};
```

identify the errors in each of the following examples and how each might be fixed:

```
(a) Derived d; d.foo(1024);
(b) void Derived::foobar() { bar = 1024; }
(c) bool Derived::bar(Base *pb)
         { return foo_bar == pb->foo_bar; }
```

Answer 15.23: As written, the class code contains two errors that must be fixed prior to answering this question. The `foo` functions in both the base and derived classes need a return type. Assume that they return `void`. The `Derived` class reuses the member name `bar` to refer both to a data member and a function member. Reusing a member name in this way is illegal. Assume that the function is neamed `bar_fcn` instead.

With these corrections to the class definitions, the errors in the use of these classes are:

(a) `d` is an object of type `Derived`. That class has a member named `foo` that takes a `string` argument. The `foo` member of the base class that takes an `int` is hidden. Assuming the caller intended to call the version in the base, one way to fix this call would be to write the call as `d.Base::foo(1024)`.

(b) The definition of `Derived::foobar`, although legal, is unlikely to execute as the author intends. We can infer that the author of this code intends to assign the integer `1024` to the base class member named `bar`. However, the derived class has its own member named `bar`. That member is type `string`. Although we cannot assign an `int` to a `string`, the `string` class does allow a `char` to be assigned to a `string`. What happens in this case is that the `int` value `1024` is converted to a `char` (which on many implementations yields an undefined value because 1024 is larger than the largest `char` value). The resulting `char` value is used to initialize the `string` member of the `Derived` part of the object. The `Base::bar` member will be unchanged.

    The best way to avoid such problems is to *never* reuse names across base and derived classes in this manner. In this case, we'd invent another name for the `string` member of `Derived` and then the definition of `Derived::foobar` can operate as is.

(c) The final example is in error because it attempts to access a protected member in its parameter. The fix for this problem depends on what these classes are supposed to represent. Perhaps the least intrusive fix would be to give `Base` a function that takes a `double` and returns a `bool` indicating whether the argument is equal to its `foo_bar` member:

```
bool Base::chk_foo_bar(double d) { return d == foo_bar; }
bool Derived::bar(Base *pb) { return chk_foo_bar(foo_bar); }
```

Exercises Section 15.5.4

Exercise 15.25: *Assume* `Derived` *inherits from* `Base` *and that* `Base` *defines each of the following functions as virtual. Assuming* `Derived` *intends to define its own version of the virtual, determine which declarations in* `Derived` *are in error and specify what's wrong.*

```
(a) Base* Base::copy(Base*);
    Base* Derived::copy(Derived*);
(b) Base* Base::copy(Base*);
    Derived* Derived::copy(Base*);
(c) ostream& Base::print(int, ostream&=cout);
    ostream& Derived::print(int, ostream&);
(d) void Base::eval() const;
    void Derived::eval();
```

Answer 15.25:

(a) The declaration in `Derived` hides rather than redefines the version inherited from `Base`. To redefine the function, the parameter type(s) must match. We might correct it by writing

```
Base* Derived::copy(Base*);
```

(b) OK: the return type of a virtual function defined in the derived class may differ from the return type defined in the base so long as the return type used in the base and derived are pointers or references to types related by inheritance. The return type in the derived version must be a pointer or reference to a type that is inherited from the return type pointed to or referred to in the base class version.

(c) Legal, but likely to cause problems to users of these classes. The problem is that the base class defines a default argument for the `ostream` parameter but the derived class does not. Users are likely to become confused because it is ok to omit the `ostream` parameter when calling the function through a base class object, pointer, or reference but they must specify an explicit `ostream` argument if calling the function through a derived type object, pointer, or reference.

(d) The declaration in `Derived` hides rather than redefines the version inherited from `Base`. The problem is that the version in the base is a `const` member function and in the derived is non a `const` member. The type of the hidden `this` parameter differs between these two functions and so the version in the derived is not a redefinition of the version in the base.

Exercises Section 15.6

Exercise 15.26: *Make your version of the* `Disc_item` *class an abstract class.*

Answer 15.26:

```
class Disc_item : public Item_base {
public:
    //  net_price is a pure virtual function
    double net_price(std::size_t) const = 0;
    Disc_item(const std::string& book = "",
              double sales_price = 0.0,
              std::size_t qty = 0, double disc_rate = 0.0):
                 Item_base(book, sales_price),
                 quantity(qty), discount(disc_rate) { }
protected:
    std::size_t quantity;  //  purchase size for discount to apply
    double discount;       //  fractional discount to apply
};
```

Exercises Section 15.7

Exercise 15.30: *Explain any discrepancy in the amount generated by the previous two programs. If there is no discrepancy, explain why there isn't one.*

Answer 15.30: Assuming `Bulk_item` objects are used in both of the previous exercises, then the sums should differ. The first program, which used a `vector<Item_base>` to hold `Bulk_item` objects, cuts the `Bulk_item` objects down to their base parts. The prices that are calculated when `net_price` is called will not reflect the discount policy.

In the second exercise, the `vector` holds pointers. When we call `net_price` the call will be resolved at runtime based on the type of the object to which the pointer points. In this case, when the pointer points at a `Bulk_item` object the proper discounts will be applied.

The sum in first exercise should be higher than the sum generated in the second exercise.

Exercises Section 15.8.2

Exercise 15.31: *Define and implement the* `clone` *operation for the limited discount class implemented in the exercises for Section 15.2.3 (p. 567).*

Answer 15.31:

```
class Lim_item : public Item_base {
public:
    Lim_item* clone() const { return new Lim_item(*this); }
    // other members as before
};
```

Exercise 15.33: *Given the version of the* `Item_base` *hierarchy that includes the* `Disc_item` *abstract base class, indicate whether the* `Disc_item` *class should implement the* `clone` *function. If not, why not? If so, why?*

Answer 15.33: There is no need for the abstract class to define its own `clone` function. The `clone` function exists to copy objects. There can be no objects of an abstract class so there is no need to define a `clone` function for the `Disc_item` class.

Exercises Section 15.8.3

Exercise 15.37: *Why did we define the* `Comp` *typedef in the* `private` *part of* `Basket`*?*

Answer 15.37: The `Comp` typedef is used strictly in the implementation of class `Basket`. Because it is part of the implementation and not part of the interface, there is no need to make it `public`.

Exercise 15.38: *Why did we define two* `private` *sections in* `Basket`*?*

Answer 15.38: In general, we prefer to put class member data and utility functions at the end of the class definition. However, unlike data or function members, type members must be seen in the class before they are used. This requirement means that the typedef must appear before we use it in the public portions of the interface. We could have defined the other private data at the beginning of the class, but for consistency with other classes we put the member data after the public interface.

Exercises Section 15.9.5

Exercise 15.40: *For the expression built in Figure 15.4 (p. 612)*

(a) List the constructors executed in processing this expression.
(b) List the calls to `display` *and to the overloaded* `<<` *operator that are made in executing* `cout << q`*.*
(c) List the calls to `eval` *made when evaluating* `q.eval`*.*

Answer 15.40:

(a) Constructors:

- The `Query` constructor that takes a `string` and the `WordQuery` constructor are executed three times to build the `WordQuery` objects from `"fiery"`, `"bird"`, and `"wind"`.
- The `BinaryQuery`, `AndQuery`, and `Query(Query_base*)` constructors are executed to create the `AndQuery` from the expression `Query("fiery") & Query("bird")`
- The `BinaryQuery`, `OrQuery`, and `Query(Query_base*)` constructors are executed to create the `OrQuery` from the previously constructed `AndQuery` and `Query("wind")`.
- Note that the code presented in the book defines the parameters to the `BinaryQuery`, `AndQuery`, and `OrQuery` constructors as plain `Query` objects rather than passing these objects as `const` references. Because the parameters are passed as copies, the copy constructor is called 12 times, six each while constructing the the `AndQuery` and `OrQuery` objects. During the construction of the `AndQuery` it is called twice to initialize the parameters to the `BinaryQuery` constructor, twice to initialize the `left` and `right` members of the `BinaryQuery` object, and two more times to initialize the parameters to the `AndQuery` object. It is called six more times when constructing the `OrQuery` object. The same four times while constructing the `BinaryQuery` and twice when calling the `OrQuery` constructor.

(b) The `<<` operator is executed five times, first on the overall `Query` object `q` and then for each operand of the `AndQuery` and the `OrQuery` objects. The `display` member is called five times: The `BinaryQuery` member is called to print the `AndQuery` and the `OrQuery` and the `WordQuery` `display` member is called to print each `WordQuery` operand.
(c) The `eval` operator is also executed five times—three times for the `WordQuery` objects and once each for the `AndQuery` and the `OrQuery`.

Chapter 16

Exercises Section 16.1.1

Exercise 16.2: *Write a function template that takes a reference to an* `ostream` *and a value, and writes the value to the stream. Call the function on at least four different types. Test your program by writing to* `cout`*, to a file, and to a* `stringstream`*.*

Answer 16.2:

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include "Sales_item.h"
using std::ostream; using std::cout; using std::endl; using std::string;
using std::ofstream; using std::ostringstream;

template <class T>
ostream &print(ostream &os, const T &val)
{
```

```
    os << val;
    return os;
}
int main()
{
    // call print to write values to cout
    print(cout, 42);                 // print an int
    print(cout, "\n");               // print a pointer to a C-style string
    string s("hello!");
    print(cout, s);                  // print a string
    print(cout, "\n");
    Sales_item item("9-999-9999");
    print(cout, item);               // print a Sales_item
    print(cout, "\n");
    // next call print to write values to a named file
    ofstream outfile("print.out");
    if (outfile) {
        print(outfile, 42);          // print an int
        print(outfile, "\n");        // print a pointer to a C-style string
        print(outfile, s);           // print a string
        print(outfile, "\n");
        print(outfile, item);        // print a Sales_item
        print(outfile, "\n");
    } else
        cout << "could not open output file" << endl;
    // finally call print to write values to an ostringstream
    ostringstream outstring;
    print(outstring, 42);          // print an int
    print(outstring, "\n");        // print a pointer to a C-style string
    print(outstring, s);           // print a string
    print(outstring, "\n");
    print(outstring, item);        // print a Sales_item
    print(outstring, "\n");
    cout << outstring.str() << endl;
    return 0;
}
```

Exercise 16.3: *When we called* `compare` *on two* `strings`*, we passed two* `string` *objects, which we initialized from string literals. What would happen if we wrote:*

```
compare("hi", "world");
```

Answer 16.3: In this case, the parameter types would be inferred to be `const char*` and a version of `compare` that compares two pointers would be instantiated. That program would compare the pointer values, not the contents of the character arrays to which the pointers pointed. It would tell us whether the pointer that pointed to "hi" was less-than or greater-than the pointer that points to "world".

Exercises Section 16.1.2

Exercise 16.5: *Define a function template to return the larger of two values.*

Answer 16.5:

```
template <class T>
T maxVal(const T& v1, const T& v2)
{
```

```
    return v1 > v2 ? v1 : v2;
}
```

Exercise 16.6: *Similar to our a simplified version of* `queue`*, write a class template named* `List` *that is a simplified version of the standard* `list` *class.*

Answer 16.6:

```
template <class Type> class List {
public:
    List();                      // default constructor
    Type &front();               // return element from head of List
    const Type &front() const;
    Type &back();                // return element at back of List
    const Type &back() const;
    void push_back(const Type &);   // add element to back of List
    void push_front(const Type &);  // add element to head of List
    void pop_back();                // remove element at back of List
    void pop_front();               // remove element at head of List
    bool empty() const;          // true if no elements in the List
private:
    // ...
};
```

Exercises Section 16.1.3

Exercise 16.7: *Explain each of the following function template definitions and identify whether any are illegal. Correct each error that you find.*

```
(a) template <class T, U, typename V> void f1(T, U, V);
(b) template <class T> T f2(int &T);
(c) inline template <class T> T foo(T, unsigned int*);
(d) template <class T> f4(T, T);
(e) typedef char Ctype;
    template <typename Ctype> Ctype f5(Ctype a);
```

Answer 16.7:

(a) Illegal: It appears that `f1` is intended to be a template function with three template type parameters. These type parameters are used to name the type of one of the function's parameters. The template parameter declaration for `U` omits the `class` or `typename` keyword. The correct declaration is:

```
template <class T, class U, typename V> void f1(T, U, V);
```

(b) Illegal: this function reuses the name `T` to define both a template type parameter and and to define the function's parameter. The function could be fixed either by changing the template parameter or the function parameter name:

```
// this version renames the template type parameter
template <class Type> Type f2(int &T);
```

(c) Illegal: the `inline` keyword is misplaced. The `inline` specifier follows the template parameter list:

```
template <class T> inline T foo(T, unsigned int*);
```

(d) Illegal: the function neglects to name a return type:

```
template <class T> ret_type f4(T, T);
```

(e) OK: Note that inside `f5` the `Ctype` typedef is hidden. Any reference to `Ctype` inside `f5` uses the template type parameter.

Exercises Section 16.1.4

Exercise 16.10: *What, if any, are the differences between a type parameter that is declared as a* `typename` *and one that is declared as a* `class`*?*

Answer 16.10: There are no differences.

Exercise 16.13: *Write a function that takes a reference to a container and prints the elements in that container. Use the container's* `size_type` *and* `size` *members to control the loop that prints the elements.*

Answer 16.13:

```
// function is valid only for vector, deque and array; it cannot be used with list
template <class T>
ostream&
printContainer(ostream &os, const T &container)
{
    typename T::size_type i = 0;
    while (i != container.size()) {
        // NOTE: uses subscript operator to access each element
        os << container[i];
        // separate all but last elment by a space
        if ( ++i != container.size())
            os << " ";
    }
    return os;
}
```

Exercise 16.14: *Rewrite the function from the previous exercise to use iterators returned from* `begin` *and* `end` *to control the loop.*

Answer 16.14:

```
// better version, can work on any container type
template <class It>
ostream&
printContainer(ostream &os, It beg, It end)
{
    while (beg != end) {
        os << *beg;

        // separate all but last elment by a space
        if (++beg != end)
            os << " ";
    }
    return os;
}
```

Exercises Section 16.1.5

Exercise 16.15: *Write a function template that can determine the size of an array.*

Answer 16.15:

```
// T is the element type N is the array dimension
template <class T, size_t N>
size_t
array_size(T (&parm)[N])
```

```
{
    return N;
}
```

Exercises Section 16.1.6

Exercise 16.17: *In the "Key Concept" box on page 95, we noted that as a matter of habit C++ programmers prefer using* `!=` *to using* `<`*. Explain the rationale for this habit.*

Answer 16.17: Requiring only `!=` but not `<` makes the code more flexible. Some types can logically express the notions of equality/inequality but do not have a logical meaning for `<` or the other relational operators. One example, which we discussed in in Section 14.3.2 (p. 520) is the `Sales_item` class. We can talk about whether two `Sales_items` are equal but it isn't clear what it means for one `Sales_item` to be greater than or less than another. Another good example are iterators other than the random access iterators. We can compare two `list` iterators to determine whether they are equal but cannot use `<` to determine wheter one `list` iterator is less-than another.

Exercises Section 16.2.1

Exercise 16.21: *Name two type conversions allowed on function arguments involved in template argument deduction.*

Answer 16.21: Conversions of a non`const` object to `const` and conversion from array or function to pointer to array element or function respectively.

Exercise 16.22: *Given the following templates*

```
template <class Type>
Type calc(const Type* array, int size);
template <class Type>
Type fcn(Type p1,Type p2;
```

which ones of the following calls, if any, are errors? Why?

```
double dobj;    float fobj;    char cobj;
int ai[5] = { 511, 16, 8, 63, 34 };

(a) calc(cobj, 'c');
(b) calc(dobj, fobj);
(c) fcn(ai, cobj);
```

Answer 16.22:

(a) Error: the first parameter to the `calc` function is a pointer and `cobj` is not a pointer.
(b) Error: `calc` takes a pointer and an `int` but `dobj` is not a pointer type.
(c) Error: The `fcn` function expects both its arguments to have the same type but the type of `ai` is a pointer and `cobj` is a `char`.

Exercises Section 16.2.2

Exercise 16.23: *The library* `max` *function takes a single type parameter. Could you call* `max` *passing it an* `int` *and a* `double`*? If so, how? If not, why not?*

Answer 16.23: We could call the `max` function on an `int` and a `double`, but we would have to supply an explicit template argument or cast one of the arguments to the type of the other. For example, assuming `i` is an `int` and `d` a `double` we could write:

```
max<double>(i, d)
```

```
max(i, static_cast<int>(d))
```

Exercise 16.25: *Use an explicit template argument to make it sensible to call* `compare` *passing two string literals.*

Answer 16.25:

```
char *cp1 = "world", *cp2 = "hi";
// illogical: compares the pointer values of cp1 and cp2
cout << compare(cp1, cp2) << endl;
// OK: compares the string contents of cp1 and cp2
cout << compare<string>(cp1, cp2) << endl;
```

Exercises Section 16.4

Exercise 16.31: *The following definition of* `List` *is incorrect. How would you fix it?*

```
template <class elemType> class ListItem;
template <class elemType> class List {
public:
    List<elemType>();
    List<elemType>(const List<elemType> &);
    List<elemType>& operator=(const List<elemType> &);
    ~List();
    void insert(ListItem *ptr, elemType value);
    ListItem *find(elemType value);
private:
    ListItem *front;
    ListItem *end;
};
```

Answer 16.31: The code is in error whenever it uses `ListItem` inside class `List` without qualifying that type by a template parameter. On the other hand, although correct, we can simplify the code by eliminating the uses of `elemType` to qualify `List`. In each case, `ListItem` should instead appear as `ListItem<elemType>`, those places that refer to `List<elemType>` can be simplified by writing `List`:

```
template <class elemType> class ListItem;
template <class elemType> class List {
public:
    // simplified by eliminating redundant use of elemType
    List();
    List(const List &);
    List& operator=(const List &);
    ~List();
    // corrected by qualifying the ListItem type by a template parameter
    void insert(ListItem<elemType> *ptr, elemType value);
    ListItem<elemType> *find(elemType value);
private:
    ListItem<elemType> *front;
    ListItem<elemType> *end;
};
```

Exercises Section 16.4.1

Exercise 16.34: *Write the member function definitions of the* `List` *class that you defined for the exercises in Section 16.1.2 (p. 628).*

Answer 16.34:

```
//  private class accessible only to List
template <class Type> class ListItem {
    friend class List<Type>;
    Type val;
    ListItem *next, prev;
    ListItem(const T& t): val(t), next(0), prev(0) { }
};
template <class Type> class List {
public:
    List(): first(0), last(0), curr(0) { } //  default constructor
    //  return element from head or tail of List
    //  unchecked operation -- results are undefined if applied to an empty List
    Type &front()              { return *first; }
    const Type &front() const  { return *first; }
    Type &back()               { return *last; }
    const Type &front() const  { return *last; }
    //  add element to front or back of List
    void push_back(const Type &);
    void push_front(const Type &)
    void pop_back();                //  remove element at back of List
    void pop_front();               //  remove element at head of List
    //  true if no elements in the List
    bool empty() const         { return last == first; }
private:
    ListItem<Type>* first;
    ListItem<Type>* last;
};
#endif
template <class T>
inline
void List<T>::push_back(const T &t)
{
    //  allocate a new element
    ListItem<T> *p = new ListItem<T>(t);
    //  check whether this is the first element on the List
    //  if so, set first to point at this element
    //  if not, hook the element into the end of the List
    if (last != 0) {
        last->next = p;     //  make the old last point at the new element
        p->prev = last;     //  make this element point back to the old last
    } else
        first = p;
    last = p; //  make the new element the new last
}
template <class T>
inline
void List<T>::push_front(const T &t)
{
    ListItem<T> *p = new ListItem<T>(t);  //  allocate a new element
    //  check whether this is the first element on the List
    //  if so, set last to point at this element
    //  if not, hook the element into the front of the List
    if (first != 0)
        p->next = first;     //  make this element point to the old first
        first->prev = p;     //  make the old first point back at the new element
    else
```

```
        last = p;
    first = p;   //  make first point to the new element
}
template <class T>
inline
void List<T>::pop_front()
{
    //  if there is only one element in the List reset both first and last
    ListItem<T> *p = first;   //  remember the current element so we can free it
    //  if there's only one element, then next and prev are already 0
    if (first == last)
        first = last = 0;
    else {
        first = first->next; //  otherwise, advance first
        first->prev = 0;     //  indicate that there is no previous element
    }
    //  free the old first element
    delete p;
}
template <class T>
inline
void List<T>::pop_back()
{
    //  if there is only one element in the List reset both first and last
    ListItem<T> *p = last;     //  remember the current element so we can free it
    //  if there's only one element, then next and prev are already 0
    if (first == last)
        first = last = 0;
    else {
        last = last->prev;   //  otherwise, advance first
        last->next = 0;      //  indicate that there is no further element
    }
    //  free the old last element
    delete p;
}
```

Exercise 16.35: *Write a generic version of the* `CheckedPtr` *class described in Section 14.7 (p. 526).*

Answer 16.35: Here we present the `CheckedPtr` class definition and many (but not all) the member and friend function definitions:

```
#include <cstddef>
/*
 * smart pointer: Checks access to elements throws an out_of_range
 *                exception if attempt to access a nonexistent element
 * users allocate and free the array
 */
template <class T> class CheckedPtr;
template <class T> bool operator==(const CheckedPtr<T>&, const CheckedPtr<T>&);
template <class T> bool operator!=(const CheckedPtr<T>&, const CheckedPtr<T>&);
template <class T> bool operator<(const CheckedPtr<T>&, const CheckedPtr<T>&);
template <class T> bool operator<=(const CheckedPtr<T>&, const CheckedPtr<T>&);
template <class T> bool operator>(const CheckedPtr<T>&, const CheckedPtr<T>&);
template <class T> bool operator>=(const CheckedPtr<T>&, const CheckedPtr<T>&);
template <class T> class CheckedPtr {
    friend bool operator==<T>(const CheckedPtr<T>&, const CheckedPtr<T>&);
```

```
    friend bool operator!=<T>(const CheckedPtr<T>&, const CheckedPtr<T>&);
    friend bool operator< <T>(const CheckedPtr<T>&, const CheckedPtr<T>&);
    friend bool operator<=<T>(const CheckedPtr<T>&, const CheckedPtr<T>&);
    friend bool operator> <T>(const CheckedPtr<T>&, const CheckedPtr<T>&);
    friend bool operator>=<T>(const CheckedPtr<T>&, const CheckedPtr<T>&);
public:
    // no default constructor; CheckedPtrs must be bound to an object
    CheckedPtr(int *b, int *e): beg(b), end(e), curr(b) { }
    // subscript operator
    T &operator[](const std::size_t);
    const T &operator[](const std::size_t) const;
    // dereference operator
    T& operator*();
    const T& operator*() const;

    // other members as before
    // increment and decrement
    CheckedPtr<T> operator++(int);      // postfix operators
    CheckedPtr<T> operator--(int);
    // other members as before
    CheckedPtr<T>& operator++();        // prefix operators
    CheckedPtr<T>& operator--();
    // other members as before
private:
    T* beg;     // pointer to beginning of the array
    T* end;     // one past the end of the array
    T* curr;    // current position within the array
};
#include "CheckedPtr.cc"

// CheckedPtr.cc
#include <iostream>
#include <stdexcept>
// two CheckedPtrs are equal if they point to the same place
// in the same array
template <class T>
bool operator==(const CheckedPtr<T> &lhs, const CheckedPtr<T> &rhs)
{
    return lhs.beg == rhs.beg
            && lhs.end == rhs.end
            && lhs.curr == rhs.curr;
}
// relational operators only apply if the CheckedPtrs refer to the same array
// If not, throw an exception
template <class T>
bool operator<(const CheckedPtr<T> &lhs, const CheckedPtr<T> &rhs)
{
    if (lhs.beg != rhs.beg || lhs.end != rhs.end)
        throw std::out_of_range("attempt to compare incomensurate CheckedPtrs");
    return lhs.curr < rhs.curr;
}
/*
 * each of the other relationals looks the same:
 * First check that the arrays are the same, throwing exception if not
 * Next apply the appropriate operation.
 * <= shown here, others left for the reader
 */
```

```
template <class T>
bool operator<=(const CheckedPtr<T> &lhs, const CheckedPtr<T> &rhs)
{
    if (lhs.beg != rhs.beg || lhs.end != rhs.end)
        throw std::out_of_range("attempt to compare incomensurate CheckedPtrs");
    return lhs.curr <= rhs.curr;
}
template <class T>
T &CheckedPtr<T>::operator[](const size_t index)
{
    // check whether index greater than or equal to the number of elements
    if (index >= end - beg)
        throw std::out_of_range("dereference past the end");
    return beg[index];
}
template <class T>
const T &CheckedPtr<T>::operator[](const size_t index) const
{
    // check whether index greater than or equal to the number of elements
    if (index >= end - beg)
        throw std::out_of_range("dereference past the end");
    return beg[index];
}
template <class T>
CheckedPtr<T> CheckedPtr<T>::operator--(int)
{
    // no check needed here, the call to prefix decrement will do the check
    CheckedPtr ret(*this);  // save current value
    --*this;                // move backward one element and check
    return ret;             // return saved state
}
// prefix: return reference to incremented/decremented object
template <class T>
CheckedPtr<T>& CheckedPtr<T>::operator++()
{
    if (curr == end)
        throw std::out_of_range
              ("increment past the end of CheckedPtr");
    ++curr;                 // advance current state
    return *this;
}
```

Exercises Section 16.4.2

Exercise 16.36: *Explain what instantiations, if any, are caused by each labeled statement.*

```
template <class T> class Stack { };
void f1(Stack<char>);                   // (a)
class Exercise {
    Stack<double> &rsd;                 // (b)
    Stack<int>    si;                   // (c)
};
int main() {
    Stack<char> *sc;                    // (d)
    f1(*sc);                            // (e)
    int iObj = sizeof(Stack< string >); // (f)
}
```

Answer 16.36:

(a) No instantiations: This statement is a declaration of a function and causes no storage to be allocated, nor any types to be instantiated.
(b) No instantiation: Using a template inside another class does not cause the template to be instantiated until an object of the enclosing type (e.g. `Exercise`) is defined. When an `Exercise` object is defined, then a `Stack<double>` will also be instantiated.
(c) No instantiation: When an object of type `Exercise` is defined, then a `Stack<int>` will also be instantiated.
(d) No instantiation: A pointer declaration does not cause a type to be instantiated. Only when the pointer is used is the type instantiated.
(e) Calling `f1` passes an object of type `Stack<char>`, which causes `Stack<char>` to be instantiated.
(f) Taking `sizeof` a class requires that the layout of the class be known. Hence, this statement instantiates `Stack<string>`.

Exercises Section 16.4.4

Exercise 16.41: *The friend declaration for* `operator<<` *in class* `Queue` *was*

```
friend std::ostream&
operator<< <Type> (std::ostream&, const Queue<Type>&);
```

What would be the effect of writing the `Queue` *parameter as* `const Queue&` *rather than* `const Queue<Type>&`*?*

Answer 16.41: Rewriting the friend declaration that way would be a syntax error. We can omit the template parameter(s) of a class type only in member declarations of that class. A friend isn't a member, so any uses of a template class must explicitly state the template parameters.

Exercise 16.42: *Write an input operator that reads an* `istream` *and puts the values it reads into a* `Queue`*.*

Answer 16.42: The input operator reads a value of the element type of the `Queue` from an `istream` and adds the value it read to the end of the `Queue`:

```
template <class T>
std::istream& operator>>(std::istream &is, Queue<T> &queue)
{
    // read a value of type T from the istream
    T val;
    is >> val;
    // if the read is successful put the value onto the back of queue
    if (is)
        queue.push_back(val);
    return is;
}
```

It is worth noting that because the input operator uses only operations in the public interface of `Queue`, there is no need to make the input operator a friend to class `Queue`.

Exercises Section 16.4.6

Exercise 16.43: *Add the* `assign` *member and a constructor that takes a pair of iterators to your* `List` *class.*

Answer 16.43:

```
template <class Type> class List {
public:
    // replace existing list by values in the input range
```

```
    template <class Iter> void assign(Iter, Iter);
private:
    // copy elements from input range into this List
    template <class Iter> void copy_elems(Iter, Iter);
    // rest of List class as before
};
template <class T> template <class Iter>
void List<T>::assign(Iter beg, Iter end)
{
    free_list();           // remove existing elements in this List
    copy_elems(beg, end); // copy elements from the input range
}
template <class T> template <class Iter>
void List<T>::copy_elems(Iter beg, Iter end)
{
    while (beg != end)
        push_back(*beg++);
}
```

Exercises Section 16.5.1

Exercise 16.46: *Explain what happens when an object of type* `Handle` *is copied.*

Answer 16.46: When a `Handle` is copied, the underlying pointer value is also copied, so that the new `Handle` and the original `Handle` both point to the same object. A use count is also incremented so that the underlying object is deleted only when the last `Handle` is destroyed.

Exercise 16.47: *What, if any, restrictions does* `Handle` *place on the types used to instantiate an actual* `Handle` *class.*

Answer 16.47: The `Handle` class expects that it is managing a simple object, not an array. This assumption is reflected in the fact that the pointer is destroyed using `delete`, not `delete[]`. Furthermore, because the pointer is deleted, if `T` is a type in an inheritance hierarchy, that type must have a virtual destructor.

Exercises Section 16.5.2

Exercise 16.51: *Rewrite the* `Query` *class from Section 15.9.4 (p. 613) to use the generic* `Handle` *class. Note that you will need to make the* `Handle` *a friend of the* `Query_base` *class to let it access the* `Query_base` *destructor. List and explain all other changes you made to get the programs to work.*

Answer 16.51:

1. Add an `#include` directive to include `Handle.h`
2. Add a `friend` declaration to the `Query_base` class to allow `Handle<Query_base>` to access the `private` members `eval` and `display`.
3. Replace the `Query_base*` member of class `Query` by a `Handle<Query_base>` member and rewrite the `Query` operations (`eval` and `display`) to forward through this `Handle` rather than through the `Query_base*` pointer.
4. Change the `Query` constructor that takes a `Query_base*` to use its pointer parameter to initialize the object's `Handle<Query_base>` member. Similarly, the `Query` constructor that takes a `string` allocates a new `WordQuery` and binds a `Handle<Query_base>` to the resulting pointer. Neither `Query` constructor needs to directly manipulate a use count. The use counting is managed by the `Handle` class.

5. Eliminate the `Query` copy control and associated utility functions. The `Handle` class manages copy control so the `Query` class can use the synthesized copy control functions.

6. No changes are needed in any of the `Query_base` derived classes. Nor are any changes required in the operations that use `Query` objects.

Exercises Section 16.6.1

Exercise 16.52: *Define a function template* `count` *to count the number of occurrences of some value in a* `vector`*.*

Answer 16.52:

```
#include <cstddef>
// we'll name our function Count to avoid conflict with the library count algorithm
template <class Iter, class T>
std::size_t Count(Iter beg, Iter end, const T &val)
{
    std::size_t cnt = 0;
    while (beg != end)
        if (*beg++ == val)
            ++cnt;
    return cnt;
}
```

Exercise 16.53: *Write a program to call the* `count` *function defined in the previous exercise passing it first a* `vector` *of* `double`*s, then a* `vector` *of* `int`*s, and finally a* `vector` *of* `char`*s.*

Answer 16.53:

```
#include <vector>
#include <string>
#include <iostream>
using std::vector; using std::string;
using std::cout; using std::endl;
int main()
{
    vector<int> vi;
    for (vector<int>::size_type i = 0; i != 10; ++i)
        vi.push_back(i);
    vector<double> vd(vi.begin(), vi.end());
    vector<char> vc(vi.begin(), vi.end());  // hack alert!
    cout << Count(vi.begin(), vi.end(), 42) << endl;  // prints 0
    cout << Count(vi.begin(), vi.end(), 0) << endl;   // prints 1
    cout << Count(vi.begin(), vi.end(), 3.14) << endl;  // prints 0
    cout << Count(vi.begin(), vi.end(), 3.0) << endl;   // prints 1
    cout << Count(vi.begin(), vi.end(), 'a') << endl;  // prints 0
    cout << Count(vi.begin(), vi.end(), '\0') << endl; // prints 1
    return 0;
}
```

Exercises Section 16.6.2

Exercise 16.56: *We explained the generic behavior of* `Queue` *if it is not specialized for* `const char*`*. Using the generic* `Queue` *template, explain what happens in the following code:*

```
Queue<const char*> q1;
q1.push("hi"); q1.push("bye"); q1.push("world");
Queue<const char*> q2(q1);  // q2 is a copy of q1

Queue<const char*> q3;      // empty Queue
q1 = q3;
```

In particular, say what the values of `q1` *and* `q2` *are after the initialization of* `q2` *and after the assignment to* `q3`*.*

Answer 16.56: This code begins by populating `q1` with three elements, each of which is a pointer to a character string literal.

It then makes a copy of this `Queue` in `q2`. After the definition of `q2`, both `q1` and `q2` hold 3 elements and those elements point to the same string literals. That is, the element at the head of `q1` and at the head of `q2` both point to the same memory location; they hold the same pointer value.

Next an empty `Queue` named `q3` is defined and that empty `Queue` is assigned to `q1`. The `Queue` assignment operator first frees the elements to which `q1` refers. It then copies the elements from `q3` into `q1`. There being no elements in `q3`, after the assignment both `q1` and `q3` are empty. The `q2` `Queue` is unchanged—it still holds 3 elements and those elements point to the string literals initially placed into `q1`.

Exercises Section 16.6.3

Exercise 16.59: *If we go the route of specializing only the* `push` *function, what value is returned by* `front` *for a* `Queue` *of C-style character strings?*

Answer 16.59: If we specialize only the `push` function, then `front` returns the pointer held in the `Queue`. It would be possible for the user to use that pointer to change the value to which the `Queue` points.

Exercises Section 16.7

Exercise 16.63: *For each of the following calls, list the candidate and viable functions. Indicate whether the call is valid and if so which function is called.*

```
template <class T> T calc(T, T);
double calc(double, double);
template <> char calc<char>(char, char);
int ival; double dval; float fd;
calc(0, ival);          calc(0.25, dval);
calc(0, fd);            calc (0, 'J');
}
```

Answer 16.63: In each call, all three `calc` functions are in the candidate set. In none of these calls is the specialization viable. To be viable, the call would have to match the type of the specialization exactly. None of the calls provides two `char` arguments, so we can ignore that function in analyzing each call in more detail.

> `calc(0, ival)` calls the template function with `T` bound to `int`. In this case, the template function and the ordinary function are viable. The template function is viable because the argument types match; to call the function template, both arguments must have the same type. The ordinary function is also viable because there is a conversion from the argument type (`int`) to the parameter type (`double`). However, it would require a conversion to call the ordinary `calc` function (converting the `int` arguments to match the `double` parameters). The template instantiated with `T` as `int` is an exact

match for this call.

`calc(0.25, dval)` calls the ordinary function taking two `doubles`. In this case both the template and ordinary functions are again viable: The argument types are the same, which means that the template function is viable. The ordinary function is viable, and indeed, the argument types and parameter types match exactly. Both the ordinary function and the function template instantiated with `double` provide exact match for this call. By the third rule of function matching we can discard the template instantiation, which leaves a single best match for this call, which is the ordinary function taking two `doubles`.

`calc(0, fd)` calls the ordinary function taking two `doubles`. In this call the function template is not viable—the argument types differ and conversions are not allowed. The ordinary `calc` function is viable because we can convert the `int` and `float` arguments to `double`, which is the type of the parameter. Because there is only one viable function, this call matches that function, the ordinary function taking two `doubles`.

`calc (0, 'J')` calls the ordinary function taking two `doubles`. Again, the function template is not viable for this call because the argument types do not match exactly. The argument types, `int` and `char`, can be converted to match the ordinary function that takes two `doubles` making it the only viable, and hence the selected, function.

Chapter 17

Exercises Section 17.1.1

Exercise 17.1: *What is the type of the exception object in the following* `throws`*:*

```
(a) range_error r("error"); (b) exception *p = &r;
    throw r;                    throw *p;
```

Answer 17.1: In the first example the exception object has type `range_error`. In the second, the type is `exception`. The type of the exception object is the static type of the thrown object. In the second throw, the type of the object to which the pointer points differs from the static type of the pointer. The fact that the types differ is irrelevant—the type of the exception object follows the static type of the object. In the second expression `p` is a pointer to `exception` so the exception object has type `exception*`.

Exercises Section 17.1.3

Exercise 17.3: *Explain why this* `try` *block is incorrect. Correct it.*

```
try {
    // use of the C++ standard library
} catch(exception) {
    // ...
} catch(const runtime_error &re) {
    // ...
} catch(overflow_error eobj) { /* ... */ }
```

Answer 17.3: The problem is in the ordering of the `catch` clauses. When finding a `catch` clause, the first matching `catch` is selected. In this case, the first `catch` catches the base-class type of the subsequent catches. `catch` clauses should be organized from most specific to least specific:

```
try {
    // use of the C++ standard library
} catch(overflow_error eobj) {
    // ...
} catch(const runtime_error &re) {
```

```
    // ...
} catch(exception) { /* ...  */ }
```

Exercises Section 17.1.5

Exercise 17.5: *Given the following exception types and* `catch` *clauses, write a* `throw` *expression that creates an exception object that could be caught by each* `catch` *clause.*

```
(a) class exceptionType { };
    catch(exceptionType *pet) { }
(b) catch(...) { }
(c) enum mathErr { overflow, underflow, zeroDivide };
    catch(mathErr &ref) { }
(d) typedef int EXCPTYPE;
    catch(EXCPTYPE) { }
```

Answer 17.5:

(a) This `catch` expects to recieve a pointer to an object of type `exceptionType`:

```
static exceptionType e_obj;    // This object must still exist at the catch point!
throw &e_obj;
```

(b) The type of the object thrown doesn't matter in this case. This `catch` clause can be matched by an exception object of any type.

(c) This `catch` matches any object of the enumerated type `mathErr`:

```
throw underflow;   // creates a mathErr object initialized to underflow
```

(d) A typedef is a synonym for a given type, not a new type. This `catch` will match an exception object of type `EXCPTYPE` or type `int`.

```
EXCPTYPE e_obj = 42;
throw e_obj;

// alternative, but less clear
throw 42;    // type of 42 is int
```

Throwing an `int` makes it less clear that we intend to throw an object that represents whatever abstraction was intended by the definition of `EXCPTYPE`.

Exercises Section 17.1.8

Exercise 17.7: *There are two ways to make the previous code exception-safe. Describe them and implement them.*

Answer 17.7: The easiest way to make this class exception safe is to use a `vector` in place of the dynamically allocated array:

```
void exercise(int *b, int *e)
{
    vector<int> v(b, e);
    // allocate equivalent vector with v.size elements
    vector<int> v2(v.size());
    ifstream in("ints");
    // exception occurs here
    // ...
}
```

A second approach would be to define a simple `struct` to hold the pointer to the dynamically allocated memory. We'll give the `struct` a constructor that will allocate the underlying array and a destructor to free it. And, we'll make the copy constructor and assignment operator private, so that we don't need to worry about managing multiple copies of the same array. Because we're inventing this class only to encapsulate the memory allocation, we'll make the pointer to the array public, which gives the code in `exercise` the same access to `p` as it had in the original:

```
// class to wrap memory allocation so that deallocation will be automatic
struct DynArray {
    DynArray(size_t n): p(new int[n]) { }
    ~DynArray() { delete [] p; }
    int *p;    // public so users can access the array directly as before
private:
    DynArray(const DynArray&);   // not implemented, so no copies can be made
    DynArray& operator=(const DynArray&);
};
void exercise(int *b, int *e)
{
    vector<int> v(b, e);
    // DynArray will be automatically freed if exception occurs
    DynArray arr(v.size());
    ifstream in("ints");
    // exception occurs here
    // ...
}
```

Exercises Section 17.1.9

Exercise 17.9: *Assuming* `ps` *is a pointer to* `string`*, what is the difference, if any, between the following two invocations of* `assign` *(Section 9.6.2, p. 339)? Which do you think is preferable? Why?*

```
(a) ps.get()->assign("Danny");   (b) ps->assign("Danny");
```

Answer 17.9: This exercise should assume that `ps` is an `auto_ptr<string>`. If it were a pointer to `string`, then `ps.get()` would be a compile time error—pointers do not have any members.

Assuming the question is phrased correctly, then the answer is that the two forms are equivalent. In the first, we call the `auto_ptr` member `get` to obtain a pointer and then call the `string` `assign` member on the object to which `ps` points through that pointer. In the second we obtain the pointer through the overloaded `auto_ptr` arrow operator.

Our preference is for the second example, `ps->assign("Danny")`. We prefer this approach largely because it is shorter. Also, it avoids using `get`. The `get` operation returns the underlying pointer value and so can be dangerous to use. By avoiding it, we don't have to think about whether this particular usage is safe.

However, neither form checks whether the `auto_ptr` is actually bound to a `string` pointer. If there is any uncertainty about whether `ps` is bound to a `string`, a better form would be to write:

```
if (ps.get())
    ps->assign("Danny");
```

This version of the code first checks that the `auto_ptr` is bound. Only after we know that it is safe to use the pointer do we do so.

Exercises Section 17.1.11

Exercise 17.11: *Which, if either, of the following initializations is in error? Why?*

```
void example() throw(string);
(a) void (*pf1)() = example;
(b) void (*pf2)() throw() = example;
```

Answer 17.11: The initialization of pf1 is ok—it makes no promises about which, if any, exceptions might be thrown so the user of pf1 must be ready to deal with any exception when making a call through pf1. The fact that the function to which it points actually says it will only throw an exception with type string doesn't matter. The function pointer specification is less restrictive than the function to which it points which means that the initialization is safe.

The second initializer is in error. This pointer promises that the function to which it points will throw no exceptions. However, example might throw an exception of type string. The promise by pf2 not to throw could be violated if the initialization from example were made. Hence, the initialization of pf2 is not allowed.

Exercises Section 17.2.1

Exercise 17.14: *Define* Sales_item *and its operators inside the* Bookstore *namespace. Define the addition operator to throw an exception.*

Answer 17.14: Our version defines both the exception classes and a simplified version of the Sales_item in the Bookstore namespace in a single file:

```
#include <stdexcept>
namespace Bookstore {
    // hypothetical exception classes for a bookstore application
    class out_of_stock: public std::runtime_error {
    public:
        explicit out_of_stock(const std::string &s):
                           std::runtime_error(s) { }
    };

    class isbn_mismatch: public std::logic_error {
    public:
        explicit isbn_mismatch(const std::string &s):
                              std::logic_error(s) { }
        isbn_mismatch(const std::string &s,
            const std::string &lhs, const std::string &rhs):
            std::logic_error(s), left(lhs), right(rhs) { }
        const std::string left, right;
        virtual ~isbn_mismatch() throw() { }
    };
    class Sales_item {
    friend std::istream& operator>>(std::istream&, Sales_item&);
    friend std::ostream& operator<<(std::ostream&, const Sales_item&);
    public:
        Sales_item(const std::string &book = " "):
                  isbn(book), units_sold(0), revenue(0.0) { }
        Sales_item(std::istream &is) { is >> *this; }
        // operations on Sales_item objects
        // member binary operator: left-hand operand bound to implicit this pointer
        Sales_item& operator+=(const Sales_item&);
        double avg_price() const { return revenue / units_sold; }
        bool same_isbn(const Sales_item &rhs) const
            { return (isbn == rhs.isbn); }
    // private members as before
    private:
        std::string isbn;
        int units_sold;
        double revenue;

    };    // end of class Sales_item
```

```
    Sales_item &
    Sales_item::operator+=(const Sales_item &rhs)
    {
        // if isbns match do the addition, otherwise throw exception
        if (same_isbn(rhs)) {
            units_sold += rhs.units_sold;
            revenue += rhs.revenue;
        } else
            throw isbn_mismatch("addition of differing isbns: ",
                                isbn, rhs.isbn);

        return *this;
    }
    // nonmember binary operator: must declare a parameter for each operand
    Sales_item
    operator+(const Sales_item &lhs, const Sales_item &rhs)
    {
        Sales_item ret(lhs);
        ret += rhs;
        return ret;
    }
}
```

Exercises Section 17.2.2

Exercise 17.17: *Over the course of this primer, we defined two different classes named* `Sales_item`*: the initial simple class defined and used in Part I, and the handle class defined in Section 15.8.1 that interfaced to the* `Item_base` *inheritance hierarchy. Define two namespaces nested inside the* `cplusplus_primer` *namespace that could be used to distinguish these two class definitions.*

Answer 17.17:

```
namespace cplusplus_primer {
    namespace part1 {
        class Sales_item {
            // definition of Sales_item from part1 goes here
        };
    }
    namespace part4 {
        class Sales_item {
            // version of Sales_item defined as handle
            // on the Item_base hierarchy goes here
        };
    }
}
```

Exercises Section 17.2.3

Exercise 17.19: *Suppose we have the following declaration of the* `operator*` *that is a member of the nested namespace* `cplusplus_primer::MatrixLib`*:*

```
namespace cplusplus_primer {
    namespace MatrixLib {
        class matrix {  /* ... */  };
        matrix operator*
               (const matrix &, const matrix &);
        // ...
    }
}
```

How would you define this operator in global scope? Provide only the prototype for the operator's definition.

Answer 17.19:

```
// function prototype for a definition of operator* at global scope
cplusplus_primer::MatrixLib::matrix
cplusplus_primer::MatrixLib::operator*(const cplusplus_primer::MatrixLib::matrix &,
                     const cplusplus_primer::MatrixLib::matrix &);
// more convenient definition would first define a typedef
typedef cplusplus_primer::MatrixLib::matrix matrix;
matrix cplusplus_primer::MatrixLib::operator*(const matrix&, const matrix&);
```

Exercises Section 17.2.4

Exercise 17.21: *Consider the following code sample:*

```
namespace Exercise {
    int ivar = 0;
    double dvar = 0;
    const int limit = 1000;
}
int ivar = 0;
// position 1
void manip() {
    // position 2
    double dvar = 3.1416;
    int iobj = limit + 1;
    ++ivar;
    ++::ivar;
}
```

What are the effects of the declarations and expressions in this code sample if `using` *declarations for all the members of namespace* `Exercise` *are located at the location labeled position 1? At position 2 instead? Now answer the same question but replace the* `using` *declarations with a* `using` *directive for namespace* `Exercise`*.*

Answer 17.21: Placing `using` declarations for all the members of namespace `Exercise` at the location indicated by the "position 1" comment makes these names available from that point in the program. Inside `manip`, the declaration of `dvar` hides the name `dvar` from `Exercise`. The reference to `limit` is a reference to `Exercise::limit`. The `using` declaration for `ivar` is a compile-time error. The `using` declaration occurs at global scope and follows the global definition of `ivar`. This `using` declaration attempts to reuse the same name `ivar` to refer to two different objects (the global `ivar` and the `ivar` object that is defined inside the `Exercise` namespace. The compiler should complain about a duplicate definition.

If the `using` declarations are moved to the location indicated by the "position 2" comment, then the declaration of `dvar` inside `manip` is an error. It is treated as a redeclaration of the name `dvar` that was made in the using declaration. As in the initial case, the use of the name limit is a use of the name from the Exercise namespace. The treatment of ivar differs from what happened when the using declaration occurred at "position 1." In this case, the names from Exercise are local to manip so there is no duplication between Exercise::ivar and the global ivar. Instead, the ivar inside manip hides the global ivar. Thus, the expression ++ivar uses the name ivar from the Exercise namespace. The second use of the name ++::ivar explicitly asks for the name from global scope.

Next we'll look at the effect of providing a using directive for the Exercise namespace instead of using declarations for the names in Exercise. In the first case, we assume the using directive appeared at the position indicated by the "position 1" comment. No error is generated at the point of the using directive. However, when the compiler attempts to translate the expression ++ivar it will generate an ambiguity error. The using directive injects the names from Exercise into the global namespace. There is no way to determine whether ++ivar should increment the global variable named ivar or the ivar that is a member of the Exercise namespace.

If we move the using directive inside manip, the ambiguity on ivar remains. The names from the namespace are injected into the nearest scope that encloses both the namespace and the scope in which the using directive occurs. That scope is the global scope. The global scope itself defines an object named ivar and so the expression ++ivar is ambiguous. As before ++::ivar is fine because it explicitly refers to the object defined at the global scope.

Exercises Section 17.2.6

Exercise 17.22: *Given the following code, determine which function, if any, matches the call to* compute*. List the candidate and viable functions. What type conversion sequence, if any, is applied to the argument to match the parameter in each viable function?*

```
namespace primerLib {
    void compute();
    void compute(const void *);
}
using primerLib::compute;
void compute(int);
void compute(double, double = 3.4);
void compute(char*, char* = 0);
int main()
{
    compute(0);
    return 0;
}
```

What would happen if the using *declaration were located in* main *before the call to* compute*? Answer the same questions as before.*

Answer 17.22: Given the code as written in the exercise, the function that is called is ::compute(int).
The viable functions are

```
::compute(int)

::compute(double, double)

::compute(char*, char*)

primerLib::compute(const void *)
```

The function primerLib::compute() is not viable because it takes no arguments and the call contains one argument. The functions ::compute(double, double) and ::compute(char*, char*) are viable even though they take two arguments because the declarations for these functions define a default argument, allowing them to be called with a single argument.

The call resolves to `::compute(int)` because there is no conversion required—the argument is an `int`, which is an exact match for the parameter type. To call `primerLib::compute(const void*)`, the argument would have to be converted to `const void*`. To call the two argument functions, the argument would have to be converted to `double` or `char*` respectively. Any of these conversions is worse than the exact match on `int`.

If the `using` declaration were moved inside the `main` function, then the functions defined at global scope would be hidden by the local declarations of the `compute` members of namespace `primerLib`. In this case, the call will resolve as a call to `primerLib::compute(const void*)`. That function can be called by converting `0` to `const void*` and so the call is viable. The function `primerLib::compute()` is the only other visible `compute` function, but it is not viable because the number of parameters (none) doesn't match the number of arguments (one) in the call. There is only one viable function and that is the one that is called.

Exercises Section 17.3.1

Exercise 17.24: *Given the following class hierarchy, in which each class defines a default constructor,*

```
class A { ... };
class B : public A { ... };
class C : public B { ... };
class X { ... };
class Y { ... };
class Z : public X, public Y { ... };
class MI : public C, public Z { ... };
```

what is the order of constructor execution for the following definition?

```
MI mi;
```

Answer 17.24: The construction order is `A`, `B`, `C`, `X`, `Y`, `Z`, and finally `MI`.

Exercises Section 17.3.2

Exercise 17.25: *Given the following class hierarchy, in which each class defines a default constructor,*

```
class X { ... };
class A { ... };
class B : public A { ... };
class C : private B { ... };
class D : public X, public C { ... };
```

which, if any, of the following conversions are not permitted?

```
D *pd = new D;
(a) X *px = pd; (c) B *pb = pd;
(b) A *pa = pd; (d) C *pc = pd;
```

Answer 17.25:

(a) OK, `X` is a public base class of `D`.
(b) Error: `A` is not a public base class of `D` and so we cannot cast a pointer to `D` to a pointer to `A`. In this hierarchy, the path from `D` to `A` goes through a private base class: `D` inherits publicly from `C`, which inherits privately from `B`, which inherits publicly from `A`. Because the path from `D` to `A` goes through the private inheritance of `C` from `B`, `A` is an inaccessible base of `C` and of any class (such as `D`) inherited from `C`.
(c) Error: although `D` is publicly derived from `C`, `C` itself inherits privately from `B`, which makes `B` a private base class of `D` and hence inaccessible.
(d) OK: `D` inherits publicly from `C`.

Exercises Section 17.3.2

Exercise 17.27: *Assume we have two base classes,* `Base1` *and* `Base2`*, each of which defines a virtual member named* `print` *and a virtual destructor. From these base classes we derive the following classes each of which redefines the* `print` *function:*

```
class D1 : public Base1 { /* ... */ };
class D2 : public Base2 { /* ... */ };
class MI : public D1, public D2 { /* ... */ };
```

Using the following pointers determine which function is used in each call:

```
Base1 *pb1 = new MI; Base2 *pb2 = new MI;
D1 *pd1 = new MI; D2 *pd2 = new MI;

(a) pb1->print();  (b) pd1->print();  (c) pd2->print();
(d) delete pb2;    (e) delete pd1;    (f) delete pd2;
```

Answer 17.27: In this code, all four pointers point to an `MI` object. Each class defines its own (virtual) version of the `print` function, so lookup through any of these pointers will find the `print` function. Because each object is an `MI` at runtime, each of the calls to `print` calls the version defined by `MI`.

The anlysis for the `delete` expressions is similar. In each case we are deleting an object of type `MI`, but doing so through a pointer to one of the base classes of `MI`. Because the destructors are virtual, the destructor that is invoked is the one for the dynamic type of the object to which the pointer points. The objects are destroyed in reverse order from which they were constructed: `MI`, `D2`, `Base2`, `D1`, `Base1`.

Exercises Section 17.3.4

Exercise 17.29: *Given the class hierarchy in the box on (p. 739) and the following* `MI::foo` *member function skeleton,*

```
int ival;
double dval;
void MI::foo(double dval) { int id; /* ...  */ }
```

(a) identify the member names visible from within `MI`*. Are there any names visible from more than one base class?*
(b) identify the set of members visible from within `MI::foo`*.*

Answer 17.29:

(a) The data members visible from within `MI` are:

- `ival` and `dvec` defined inside the `MI` class;
- `sval` and `dval` defined inside the `Derived` class;
- `ival`, `dval`, `cval`, and `id` defined inside the `Base1` class;
- `dval` and `fval` defined inside the `Base2` class.

From the perspective of `MI`, the name `dval` is ambiguous—it appears in the hierarchy defined along both the `Derived` and `Base2` inheritance subtrees. Moreover, the name `Base1::dval` is hidden by the declaration of `dval` inside `Derived`. Similarely, the name `Base1::ival` is hidden by the member named `ival` inside `MI` itself.

In addition to these data members, `MI` also has four member functions named `print`. However, only the member `MI::print(vector<double>)` is directly accessible from `MI`. To use the versions of `print` defined in `Derived`, `Base2`, or `Base1`, a member of `MI` would have to use the scope operator to indicate which member was wanted.

(b) Inside `MI::foo`, all the members of `MI` are accessible. However, the name `dval` refers to the function's parameter—the use of `dval` as the parameter name hides the members named `dval` and hides the global variable named `dval`. To use the global `dval` or one of the members named `dval`, the code inside `foo` must use the scope operator to explicitly indicate a `dval` other than the parameter named `dval`. The member `id` inherited from `Base1` is also hidden by the local variable of that name. The global name `ival` is hidden by the member name `ival`. To access the global `ival`, the code in `MI::foo` must use the scope operator.

Exercises Section 17.3.4

Exercise 17.30: *Given the hierarchy in the box on page 739, why is this call to* `print` *an error?*

```
MI mi;
mi.print(42);
```

Revise `MI` *to allow this call to* `print` *to compile and execute correctly.*

Answer 17.30: The `print` member that is defined inside `MI` takes a `vector<double>` and the argument to this call is an `int`. That type cannot be (implicitly) converted to a `vector<double>`, so the call is in error. We might infer that the call intended to call the version of `print` defined in the base class `Base1`. However, that `print` function is hidden by the one defined inside `MI`. To call the version of `print` that takes an `int`, the call must be written as `Base1::print(42)`. We could also add a using declaration of the form:

```
using Base1::print;
```

to class `MI`. Doing so would put the `print` version from `Base1` in the scope of class `MI`. Calls such as `mi.print(42)` would then be allowed.

Exercise 17.32: *Using the class hierarchy defined in the box on page 739 and the following skeleton of the* `MI::foobar` *member function*

```
void MI::foobar(double cval)
{
    int dval;
   // exercise questions occur here ...
}
```

(a) assign to the local instance of `dval` *the sum of the* `dval` *member of* `Base1` *and the* `dval` *member of* `Derived`*.*
(b) assign the value of the last element in `MI::dvec` *to* `Base2::fval`*.*
(c) assign `cval` *from* `Base1` *to the first character in* `sval` *from* `Derived`*.*

Answer 17.32:

```
void MI::foobar(double cval)
{
    int dval;
    // exercise questions occur here ...
    // (a) add dval from Base1 to dval from Derived
    // and store result in the local variable dval
    // cast the result of the addition, because the local is an int
    dval = static_cast<int>(Base1::dval + Derived::dval);
    // (b) assign last element in dvec to fval
    // dvec and fval occur only once in the hierarchy so we can
    // use them directly
    if (!dvec.empty())
        fval = *dvec.rbegin();
    // (c) assign cval from Base1 to first character in sval from Derived
```

```
        // cval is hidden by the parameter named cval
        // sval occurs only once in the hierarchy; it is directly accessible
        if (!sval.empty())
            sval[0] = Base1::cval;
        else
            sval.push_back(Base1::cval);
    }
```

Exercises Section 17.3.6

Exercise 17.33: *Given the following class hierarchy, which inherited members can be accessed without qualification from within the* `VMI` *class? Which require qualification? Explain your reasoning.*

```
class Base {
public:
    bar(int);
protected:
    int ival;
};
class Derived1 : virtual public Base {
public:
    bar(char);
    foo(char);
protected:
    char cval;
};
class Derived2 : virtual public Base {
public:
    foo(int);
protected:
    int  ival;
    char cval;
};
class VMI : public Derived1, public Derived2 { };
```

Answer 17.33: A `VMI` object has members named: `bar`, `ival`, `foo`, and `cval`. All of these names are reused within the `VMI` hierarchy. Some hide previous members along the same inheritance subtree, others are ambiguous because they name members inherited along different subtrees:

- `bar` is directly accessible from `VMI` and refers to the member `bar` defined inside `Derived1`. This member hides the member `Base::bar`. Although `Base` is inherited along both `VMI` subtrees, the `Base` class is a virtual base class and so is shared by both subtrees. To access the member from the base, we must explicitly ask for it using the scope operator.

- `ival` is treated the same way as `bar`: The definition of `Derived2::ival` hides `Base::ival`. Because `Base` is a shared, virtual base class, there is only one `Base::ival`, which is hidden by the definition of `ival` inside `Derived2`.

- `foo` is defined in both `Derived1` and `Derived2`. These nonvirtual base classes are defined along different subtrees of `VMI`. Any unqualified reference to `foo` from a `VMI` member is ambiguous.

- `cval`, like `foo`, is defined by two nonvirtual base classes found along different inheritance paths from `VMI`. To use `cval` we must explicitly say which class member we want.

Exercises Section 17.3.7

Exercise 17.35: *Given the following class hierarchy,*

```
class Class { ... };
class Base : public Class { ... };
class Derived1 : virtual public Base { ... };
class Derived2 : virtual public Base { ... };
class MI : public Derived1,
           public Derived2 { ... };
class Final : public MI, public Class { ... };
```

(a) What is the order of constructor and destructor for the definition of a `Final` *object?*
(b) How many `Base` *subobjects are in a* `Final` *object? How many* `Class` *subobjects?*
(c) Which of the following assignments is a compile-time error?

```
Base     *pb;      Class    *pc;
MI       *pmi;     Derived2 *pd2;
(a) pb = new Class;          (c) pmi = pb;
(b) pc = new Final;          (d) pd2 = pmi;
```

Answer 17.35:

(a) A `Final` object is constructed in the following order: The shared virtual base class is constructed first, which means that the `Class` and `Base` constructors are run in that order to construct the shared `Class` subobject. Next, the `Derived1` and `Derived2` and `MI` subobjects are constructed. Then, a second, unshared `Class` subobject is created to reflect the direct inheritance of `Final` from `Class`. Finally, the `Final` part is constructed.

   A `Final` object is destroyed in reverse order: the `Final` subobject is destroyed first, followed by the (nonvirtual) `Class` subobject, then the `MI`, `Derived2`, `Derived1` subobjects, followed by the shared base class subobject `Base` and its base class `Class`.

(b) There is one (shared) `Base` subobject and two `Class` subobjects in a `Final` object. The `Class` subobjects are the one from which `Final` inherits directly and the `Class` subobject from which the shared `Base` object inherits.

(c) (a) Error: attempts to initialize a pointer to a derived class from a pointer to a base class.
   (b) Error: a `Final` object has two `Class` subobjects so the conversion from a pointer to `Final` to a pointer to its base class `Class` is ambiguous.
   (c) Error: attempts to initialize a pointer to a derived class from a pointer to a base class.
   (d) OK: assigns a pointer to a derived object to a pointer to an unambiguous base class of that derived object.

Chapter 18

Exercises Section 18.1.2

Exercise 18.1: *Implement your own version of the* `Vector` *class including versions of the* `vector` *members* `reserve` *(Section 9.4, p. 330),* `resize` *(Section 9.3.5, p. 323), and the* `const` *and nonconst subscript operators (Section 14.5, p. 522).*

Answer 18.1: To support `resize` and `reserve`, we changed the interface to `reallocate` to take an optional parameter that indicates a minimal number of elements for which to allocate space:

```
// psuedo-implementation of memory allocation strategy for a vector-like class
template <class T> class Vector {
public:
```

```
    Vector(): elements(0), first_free(0), array_end(0) { }
    void push_back(const T&);
    std::size_t size() const { return first_free - elements; }
    std::size_t capacity() const { return array_end - elements; }
    void resize(std::size_t);
    void reserve(std::size_t);
    T& operator[](std::size_t n) { return elements[n]; }
    const T& operator[](std::size_t n) const { return elements[n]; }
private:
    static std::allocator<T> alloc; // member to handle allocation
    // get more space and copy existing elements; the argument says how many
    // elements the new space should be able to handle. The default argument
    // is used to cause the allocation to use twice the current size for the new size
    void reallocate(std::size_t = 1);
    T* elements;         // pointer to first element in the array
    T* first_free;       // pointer to first free element in the array
    T* array_end;        // pointer to one past the end of the array
    // ...
};

template <class T> void Vector<T>::reserve(size_t n)
{
    if (n > capacity())
        reallocate(n);
}
template <class T> void Vector<T>::resize(size_t n)
{
    // if n less than current size, free excess elements and reset first_free
    if (size() > n) {
        while (n != size()) {
            // --first_free points to last element,
            // after the call to destroy, first_free again points to
            // the first free space on the free list
            alloc.destroy(--first_free);
        }
    } else if (size() < n) {  // need more elements
        reallocate(n);     // get space for at least n elements
                           // and copy existing elements into new space
        // allocate additional default initialized elements
        while (first_free != elements + n)
            push_back(T());
    }
}

template <class T> void Vector<T>::reallocate(size_t n)
{
    // compute size of current array and double it
    std::ptrdiff_t size = first_free - elements;
    size_t newcapacity = 2 * max(size, 1);
    // allocate greater of twice the size of the current array or the parameter n
    newcapacity = max(newcapacity, n);
    // allocate space to hold newcapacity number of elements of type T
    T* newelements = alloc.allocate(newcapacity);

    // construct copies of the existing elements in the new space
    uninitialized_copy(elements, first_free, newelements);
    // destroy the old elements in reverse order
    for (T *p = first_free; p != elements; /* empty */ )
```

```
            alloc.destroy(--p);

        // deallocate cannot be called on a 0 pointer
        if (elements)
            // return the memory that held the elements
            alloc.deallocate(elements, array_end - elements);
        // make our data structure point to the new elements
        elements = newelements;
        first_free = elements + size;
        array_end = elements + newcapacity;
    }
```

The other members are unchanged from how they were presented in the text.

Exercise 18.2: *Define a typedef that uses the corresponding pointer type as the* `iterator` *for your* `Vector`.

Answer 18.2:

```
    // psuedo-implementation of memory allocation strategy for a vector-like class
    template <class T> class Vector {
    public:
        typedef T* iterator;
        typedef const T* const_iterator;
        iterator begin() { return elements; }
        iterator end()   { return first_free; }
        const_iterator begin() const { return elements; }
        const_iterator end() const   { return first_free; }
        // other members as before
    };
```

Exercises Section 18.1.4

Exercise 18.4: *Why do you think* `construct` *is limited to using only the copy constructor for the element type?*

Answer 18.4: Because the `allocator` class is a template class. Therefore, it can make no assumptions about the types used to instantiate the `allocator` class. In particular, it cannot make assumptions about which constructors a class might offer. However, all containers require that their element types be copyable. It is reaonsable for the `allocator` class to impose the same requirement on types that use it. Hence, `construct` can assume that it is possible to copy the type with which it is instantiated.

Exercises Section 18.1.6

Exercise 18.9: *Declare members* `new` *and* `delete` *for the* `QueueItem` *class.*

Answer 18.9:

```
    class QueueItem {
    public:
        void *operator new(std::size_t);
        void operator delete(void *, std::size_t);
        // other members as defined in Chapter 16
    };
```

Exercises Section 18.1.7

Exercise 18.10: *Explain each of the following initializations. Indicate if any are errors, and if so, why.*

```
class iStack {
public:
    iStack(int capacity): stack(capacity), top(0) { }
private:
    int top;
    vector<int> stack;
};
(a) iStack *ps = new iStack(20);
(b) iStack *ps2 = new const iStack(15);
(c) iStack *ps3 = new iStack[ 100 ];
```

Answer 18.10:

(a) OK: `ps` points to a dynamically allocated `iStack` object, which is initialized with 20 value-initialized `int` elements.
(b) Error: `ps2` is a non`const` pointer to type `iStack` but the initializer is a pointer to a to a dynamically allocated `iStack` object that is `const`. That object holds 15 elements, each of which is default initialized and cannot be subsequently changed. It is not possible to initialize a plain, non`const` pointer from the address of a `const` object so the initialization fails at compile time.
(c) Error: the `new` expression intends to allocate an array of `iStack` objects. However, no element initializer is provided and the `iStack` class does not have a default constructor. As written, the `new` expression will not compile.

Exercises Section 18.2.1

Exercise 18.13: *Given the following class hierarchy in which each class defines a* `public` *default constructor and virtual destructor,*

```
class A { /* ...  */ };
class B : public A { /* ...  */ };
class C : public B { /* ...  */ };
class D : public B, public A { /* ...  */ };
```

which, if any, of the following `dynamic_cast`*s fail?*

```
(a) A *pa = new C;
    B *pb = dynamic_cast< B* >(pa);
(b) B *pb = new B;
    C *pc = dynamic_cast< C* >(pb);
(c) A *pa = new D;
    B *pb = dynamic_cast< B* >(pa);
```

Answer 18.13:

(a) OK: `pa` points to a publicly accessible base class of `C`, so we can initialize `pa` with the address of a `C` object. That object has a publicly accessible `B` part so the dynamic cast from a pointer to `A` to a pointer to `B` is safe and the cast will succeed.
(b) Fails at runtime: `pb` is a pointer to `B` and is initialized to point to a dynamically allocated `B` object. The `dynamic_cast` will compile, but fail at runtime. The pointer `pb` points to a `B` object, which has no `C` part. When we attempt to cast `pb` to a `C*` that cast will fail.
(c) Fails at compile time: `D` has two `A` parts, the one it inherits directly and the one it inherits indirectly through `B`. When we try to initialize a `pa`, which is a pointer to `A` from the address of a dynamically allocated `D` object the compiler doesn't know which `A` subobject to choose. The initialization is, therefore, ambiguous.

Exercise 18.14: *What would happen in the last conversion in the previous exercise if both* `D` *and* `B` *inherited from* `A` *as a virtual base class?*

Answer 18.14: If both `D` and `B` inherited virtually from `A`, then the initialization of `pa` would succeed. `pa` would point to the shared `A` part of a `D` object. The `dynamic_cast` would succeed at runtime—a `D` object has a publicly accessible `B` part and so the attempt to cast `pa` to a pointer to `B` would succeed.

Exercises Section 18.2.2

Exercise 18.19: *Write a* `typeid` *expression to see whether two* `Query_base` *pointers point to the same type. Now check whether that type is an* `AndQuery`*.*

Answer 18.19:

```
#include "Query.h"
#include <typeinfo>
using std::type_info;
void f(Query *op1, Query *op2)
{
     // compare the types of the pointers
     if (typeid(*op1) == typeid(*op2)) {
          // if they're equal see whether they point to an AndQuery
          if (typeid(*op1) == typeid(AndQuery)) {
               // do some processing
          }

     }
}
```

Exercises Section 18.2.4

Exercise 18.20: *Given the following class hierarchy in which each class defines a* `public` *default constructor and virtual destructor, which type name do the following statements print?*

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public B { /* ... */ };

(a) A *pa = new C;
    cout << typeid(pa).name() << endl;
(b) C cobj;
    A& ra = cobj;
    cout << typeid(&ra).name() << endl;
(c) B *px = new B;
    A& ra = *px;
    cout << typeid(ra).name() << endl;
```

Answer 18.20:

(a) The printed type indicates that `pa` is a pointer to an `A`.
(b) The printed type indicates that `&ra` is a pointer to an `A`.
(c) The printed type indicates that `ra` is a `B` object.

Exercises Section 18.3.1

Exercise 18.22: *Define the type that could represent a pointer to the* `isbn` *member of the* `Sales_item` *class.*

Answer 18.22: `string Sales_item::*`

Exercise 18.23: *Define a pointer that could point to the* `same_isbn` *member.*

Answer 18.23: `string (Sales_item::*)(const string&) const`

Exercises Section 18.3.2

Exercise 18.28: *Pointers to members may also be declared as class data members. Modify the* `Screen` *class definition to contain a pointer to a* `Screen` *member function of the same type as* `home` *and* `end`*.*

Answer 18.28: First, note that the question refers to the nonexistent member `end`. It should instead refer to `down` or one of the other added cursor movement functions that take the same (empty) parameter list and return type that the `home` function has. Given this correction to the exercise, the solution is:

```
class Screen {
public:
    typedef Screen& (Screen::*Action)();
private:
    Action default_direction;
    // other members as before
};
```

Exercise 18.30: *Provide a default argument for this parameter. Use this parameter to initialize the data member introduced in the previous exercise.*

Answer 18.30:

```
class Screen {
public:
    typedef Screen& (Screen::*Action)();
    Screen(const Action &movement = &Screen::home, index ht = 0, index wd = 0):
                default_direction(movement),
                contents(ht * wd, ' '), cursor(0),
                height(ht), width(wd) { }
    // other members as before
private:
    Action default_direction;
    std::string contents;
    index cursor;
    index height, width;
};
```

Exercise 18.31: *Provide a* `Screen` *member function to set this member.*

Answer 18.31:

```
class Screen {
public:
    typedef Screen& (Screen::*Action)();
    // function to set the default_direction member
    Action set_direction(const Action &dir) { Action old = default_direction;
                                              default_direction = dir;
                                              return old; }
    // other members as before
```

Exercises Section 18.7.3

Exercise 18.34: *Explain these declarations and indicate whether they are legal:*

```
extern "C" int compute(int *, int);
extern "C" double compute(double *, double);
```

Answer 18.34: In isolation these declarations are legal: The first declares a C function that takes a pointer to an `int` and an `int` and returns an `int`. The second declares a C function that takes a pointer to a `double` and a `double` and returns a `double`. The problem is that both declarations declare the same name. C does not support function overloading. Hence, there can be only one C function with a given name. The second declaration of `compute` is, therefore, in error.